Structural Subtyping as Parametric Polymorphism

WENHAO TANG, The University of Edinburgh, United Kingdom
DANIEL HILLERSTRÖM, Huawei Zurich Research Center, Switzerland
JAMES MCKINNA, Heriot-Watt University, United Kingdom
MICHEL STEUWER, Technische Universität Berlin, Germany and the University of Edinburgh, UK
ORNELA DARDHA, University of Glasgow, United Kingdom
RONGXIAO FU, The University of Edinburgh, United Kingdom
SAM LINDLEY, The University of Edinburgh, United Kingdom

Structural subtyping and parametric polymorphism provide a similar kind of flexibility and reusability to programmers. For example, both enable the programmer to supply a wider record as an argument to a function that expects a narrower one. However, the means by which they do so differs substantially, and the precise details of the relationship between them exists, at best, as folklore in literature.

In this paper, we systematically study the relative expressive power of structural subtyping and parametric polymorphism. We focus our investigation on establishing the extent to which parametric polymorphism, in the form of row and presence polymorphism, can encode structural subtyping for variant and record types. We base our study on various Church-style λ-calculi extended with records and variants, different forms of structural subtyping, and row and presence polymorphism.

We characterise expressiveness by exhibiting compositional translations between calculi. For each translation we prove a type preservation and operational correspondence result. We also prove a number of non-existence results. By imposing restrictions on both source and target types, we reveal further subtleties in the expressiveness landscape, the restrictions enabling otherwise impossible translations to be defined. More specifically, we prove that full subtyping cannot be encoded via polymorphism, but we show that several restricted forms of subtyping can be encoded via particular forms of polymorphism.

1 INTRODUCTION

Subtyping and parametric polymorphism offer two distinct means for writing modular and reusable code. Subtyping allows one value to be substituted for another provided that the type of the former is a subtype of that of the latter [Cardelli 1988; Reynolds 1980]. Parametric polymorphism allows functions to be defined generically over arbitrary types [Girard 1972; Reynolds 1974].

There are two main approaches to *syntactic* subtyping: nominal subtyping [Birtwistle et al. 1979] and structural subtyping [Cardelli 1984, 1988; Cardelli and Wegner 1985]. The former defines a subtyping relation as a collection of explicit constraints between named types. The latter defines a subtyping relation inductively over the structure of types. This paper is concerned with the latter. For programming languages with variant types (constructor-labelled sums) and record types (field-labelled products) it is natural to define a notion of structural subtyping. We may always treat a variant with a collection of constructors as a variant with an *extended* collection of constructors (i.e., variant subtyping is covariant). Dually, we may treat a record with a collection of fields as a record with a *restricted* collection of those fields (i.e., record subtyping is contravariant).

We can implement similar functionality to record and variant subtyping using *row polymorphism* [Rémy 1994; Wand 1987]. A *row* is a mapping from labels to types and is thus a common ingredient for defining both variants and records. Row polymorphism is a form of parametric

Authors' addresses: Wenhao Tang, The University of Edinburgh, United Kingdom, wenhao.tang@ed.ac.uk; Daniel Hillerström, Huawei Zurich Research Center, Switzerland, daniel.hillerstrom@ed.ac.uk; James McKinna, Heriot-Watt University, United Kingdom, j.mckinna@hw.ac.uk; Michel Steuwer, Technische Universität Berlin, Germany and the University of Edinburgh, UK, michel.steuwer@tu-berlin.de; Ornela Dardha, University of Glasgow, United Kingdom, ornela.dardha@glasgow.ac.uk; Rongxiao Fu, The University of Edinburgh, United Kingdom, s1742701@sms.ed.ac.uk; Sam Lindley, The University of Edinburgh, United Kingdom, sam.lindley@ed.ac.uk.

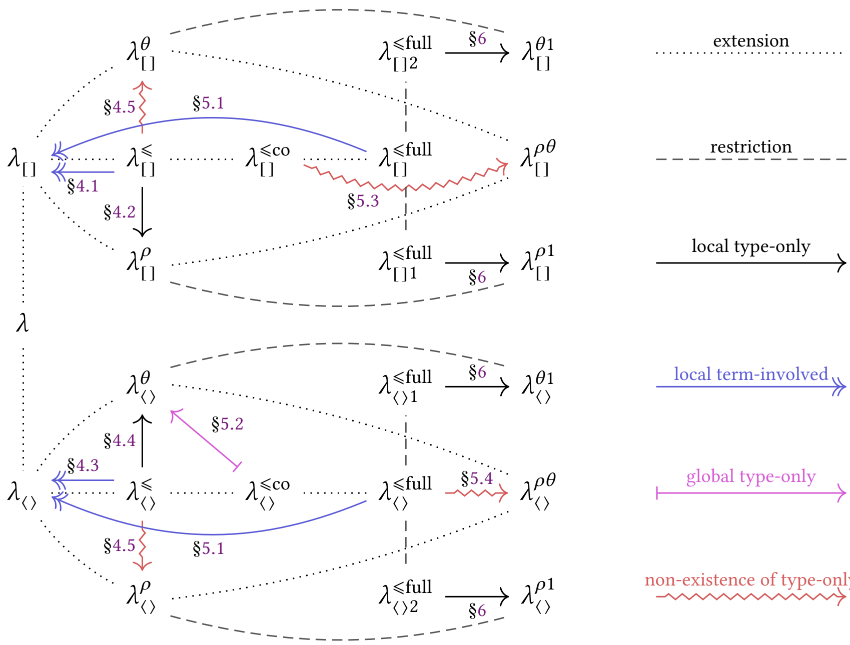
polymorphism that allows us to abstract over the extension of a row. Intuitively, by abstracting over the possible extension of a variant or record we can simulate the act of substitution realised by structural subtyping. Such intuitions are folklore, but pinning them down turns out to be surprisingly subtle. In this paper we make them precise by way of translations between a series of different core calculi enjoying type preservation and operational correspondence results as well as non-existence results. We show that though folklore intuitions are to some extent correct, exactly how they manifest in practice is remarkably dependent on what assumptions we make, and much more nuanced than we anticipated. We believe that our results are not just of theoretical interest. It is important to carefully analyse and characterise the relative expressive power of different but related features to understand the extent to which they overlap, placing the design of practical programming language on a more scientific basis.

To be clear, there is plenty of other work that hinges on inducing a subtyping relation based on generalisation (i.e. polymorphism) — and indeed this is the basis for principal types in Hindley-Milner type inference — but that this paper is about something quite different, namely encoding prior notions of structural subtyping using polymorphism. In short, principal types concern polymorphism as subtyping whereas this paper concerns subtyping as polymorphism.

In order to distil the features we are interested in down to their essence and eliminate the interference on the expressive power of other language features (such as higher-order store), we take plain Church-style call-by-name simply-typed λ-calculus (λ) as our starting point and consider the relative expressive power of minimal extensions in turn. We begin by insisting on writing explicit upcasts, type abstractions, and type applications in order to expose structural subtyping and parametric polymorphism at the term level. Later we also consider ML-style calculi, enabling new expressiveness results by exploiting the type inference for rank-1 polymorphism. For the dynamic semantics, we focus on the reduction theory generated from the β-rules, adding further β-rules for each term constructor and upcast rules for witnessing subtyping.

First we extend the simply-typed λ-calculus with variants ($\lambda_{[]}$), which we then further augment with *simple subtyping* ($\lambda_{[]}^{\leqslant}$) that only considers the subtyping relation shallowly on variant and record constructors (width subtyping), and (higher-rank) row polymorphism ($\lambda_{[]}^{\rho}$), respectively. Dually, we extend the simply-typed λ-calculus with records ($\lambda_{\langle\rangle}$), which we then further augment with simple subtyping ($\lambda_{\langle\rangle}^{\leqslant}$) and (higher-rank) *presence polymorphism* ($\lambda_{\langle\rangle}^{\theta}$) respectively. Presence polymorphism [Rémy 1994] is a kind of dual to row polymorphism that allows us to abstract over which fields are present or absent from a record independently of their potential types, supporting a restriction of a collection of record fields, similarly to record subtyping. We then consider richer extensions with strictly covariant subtyping ($\lambda_{[]}^{\leqslant\mathrm{co}}$, $\lambda_{\langle\rangle}^{\leqslant\mathrm{co}}$) which propagates the subtyping relation through strictly covariant positions, and full subtyping ($\lambda_{[]}^{\leqslant\mathrm{full}}$, $\lambda_{\langle\rangle}^{\leqslant\mathrm{full}}$) which propagates the subtyping relation through any positions. We also consider target languages with both row and presence polymorphism ($\lambda_{[]}^{\rho\theta}$, $\lambda_{\langle\rangle}^{\rho\theta}$). Our initial investigations make essential use of higher-rank polymorphism. Subsequently, we consider ML-like calculi with rank-1 row or presence polymorphism ($\lambda_{\langle\rangle}^{\rho 1}$, $\lambda_{\langle\rangle}^{\theta 1}$, $\lambda_{[]}^{\rho 1}$, $\lambda_{[]}^{\theta 1}$), which admit Hindley-Milner type inference [Damas and Milner 1982] without requirements of type annotations or explicit type abstractions and applications. The focus on rank-1 polymorphism demands a similar restriction to the calculi with subtyping ($\lambda_{\langle\rangle 1}^{\leqslant\mathrm{full}}$, $\lambda_{\langle\rangle 2}^{\leqslant\mathrm{full}}$, $\lambda_{[]1}^{\leqslant\mathrm{full}}$, $\lambda_{[]2}^{\leqslant\mathrm{full}}$), which constrains the positions where records and variants can appear in types.

In this paper, we will consider only correspondences expressed as *compositional translations* inductively defined on language constructs following Felleisen [1991]. In order to give a refined characterisation of expressiveness and usability of the type systems of different calculi, we make use of two orthogonal notions of *local* and *type-only* translations.



Extensions and restrictions go from calculi with shorter names to those with longer names (e.g. $\lambda_{[]}$ extends λ and $\lambda^{\theta 1}_{[]}$ restricts $\lambda^{\theta}_{[]}$).

Fig. 1. Overview of translations and non-existence results covered in the paper.

- A *local* translation restricts which features are translated in a non-trivial way. It provides non-trivial translations only of constructs of interest (e.g., record types, record construction and destruction, when considering record subtyping), and is homomorphic on other constructs; a *global* translation may allow any construct to have a non-trivial translation.
- A *type-only* translation restricts which features a translation can use in the target language. Every term must translate to itself modulo constructs that serve only to manipulate types (e.g., type abstraction and application); a *term-involved* translation has no such restriction.

Local translations capture the intuition that a feature can be expressed locally as a macro rather than having to be implemented by globally changing an entire program [Felleisen 1991]. Type-only translations capture the intuition that a feature can be expressed solely by adding or removing type manipulation operations (such as upcasts, type abstraction, and type application) in terms, thereby enabling a more precise comparison between the expressiveness of different type system features.

This paper gives a *precise account of the relationship between subtyping and polymorphism for records and variants*. We present relative expressiveness results by way of a series of translations between calculi, type preservation proofs, operational correspondence proofs, and non-existence proofs. The main contributions of the paper (summarised in Figure 1) are as follows.

- We present a collection of examples in order to convey the intuition behind all translations and non-existence results in Figure 1 (Section 2).
- We define a family of Church-style calculi extending lambda-calculus with variants and records, simple subtyping, (higher-rank) row polymorphism, and (higher-rank) presence polymorphism (Section 3).

- We prove that simple subtyping can be elaborated away for variants and records by way of local term-involved translations (Sections 4.1 and 4.3).
- We prove that simple subtyping can be expressed as row polymorphism for variants and presence polymorphism for records by way of local type-only translations (Sections 4.2 and 4.4).
- We prove that there exists no type-only translation of simple subtyping into presence polymorphism for variants or row polymorphism for records (Section 4.5).
- We expand our study to calculi with covariant and full subtyping and with both row- and presence-polymorphism, covering further translations and non-existence proofs (Section 5). In so doing we reveal a fundamental asymmetry between variants and records.
- We prove that if we suitably restrict types and switch to ML-style target calculi with implicit rank-1 polymorphism, then we can exploit type inference to encode full subtyping for records and variants using either row polymorphism or presence polymorphism (Section 6).
- For each translation we prove type preservation and operational correspondence results.

Sections 7.1 and 7.2 discuss extensions. Section 7.3 discusses related work. Section 7.4 concludes.

2 EXAMPLES

To illustrate the relative expressive power of subtyping and polymorphism for variants and records with a range of extensions, we give a collection of examples. These cover the intuition behind the translations and non-existence results summarised in Figure 1 and formalised later in the paper.

2.1 Simple Variant Subtyping as Row Polymorphism

We begin with variant types. Consider the following function.

$$\mathtt{getAge} = \lambda x^{[\mathsf{Age}:\mathsf{Int};\mathsf{Year}:\mathsf{Int}]}.\ \mathbf{case}\ x\ \{\mathsf{Age}\ y \mapsto y; \mathsf{Year}\ y \mapsto 2023 - y\}$$

The variant type $[\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}]$ denotes the type of variants with two constructors Age and Year each containing an Int. We cannot directly apply `getAge` to the following variant

$$\mathtt{year} = (\mathsf{Year}\ 1984)^{[\mathsf{Year}:\mathsf{Int}]}$$

as `year` and x have different types. With simple variant subtyping ($\lambda_{[\,]}^{\leqslant}$) which considers subtyping shallowly on variants, we can upcast $\mathtt{year} : [\mathsf{Year} : \mathsf{Int}]$ to the supertype $[\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}]$ which has more labels. This makes intuitive sense, as it is always safe to treat a variant with fewer constructors (Year in this case) as one with more constructors (Age and Year in this case).

$$\mathtt{getAge}\ (\mathtt{year} \rhd [\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}])$$

One advantage of subtyping is reusability: by upcasting we can apply the same `getAge` function to any value whose type is a subtype of $[\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}]$.

$$\mathtt{age} = (\mathsf{Age}\ 9)^{[\mathsf{Age}:\mathsf{Int}]}$$
$$\mathtt{getAge}\ (\mathtt{age} \rhd [\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}])$$

In a language without subtyping ($\lambda_{[\,]}$), we can simulate applying `getAge` to `year` by first deconstructing the variant using **case** and then reconstructing it at the appropriate type — a kind of generalised η-expansion on variants.

$$\mathtt{getAge}\ (\mathbf{case}\ \mathtt{year}\ \{\mathsf{Year}\ y \mapsto (\mathsf{Year}\ y)^{[\mathsf{Age}:\mathsf{Int};\mathsf{Year}:\mathsf{Int}]}\})$$

This is the essence of the translation $\lambda_{[\,]}^{\leqslant} \twoheadrightarrow \lambda_{[\,]}$ in Section 4.1. The translation is *local* in the sense that it only requires us to transform the parts of the program that relate to variants (as opposed to the entire program). However, it still comes at a cost. The deconstruction and reconstruction of variants adds extra computation that was not present in the original program.

Can we achieve the same expressive power of subtyping without non-trivial term de- and re-construction? Yes we can! Row polymorphism ($\lambda^{\rho}_{[]}$) allows us to rewrite `year` with a type compatible (via row-variable substitution) with any variant type containing $\mathsf{Year} : \mathsf{Int}$ and additional cases. [1]

$$\texttt{year}' = \Lambda\rho.\ (\mathsf{Year}\ 1984)^{[\mathsf{Year}:\mathsf{Int};\rho]}$$

As before, the translation to `year′` also adds new term syntax. However, the only additional syntax required by this translation involves type abstraction and type application; in other words the program is unchanged up to type erasure. Thus we categorise it as a *type-only* translation as opposed to the previous one which we say is *term-involved*. We can instantiate ρ with $(\mathsf{Age} : \mathsf{Int})$ when applying `getAge` to it. The parameter type of `getAge` must also be translated to a row-polymorphic type, which requires higher-rank polymorphism. Moreover, we re-abstract over `year′` after instantiation to make it polymorphic again.

$$\begin{array}{l}\texttt{getAge}' = \lambda x^{\forall\rho.[\mathsf{Age}:\mathsf{Int};\mathsf{Year}:\mathsf{Int};\rho]}.\ \mathbf{case}\ (x\ \cdot)\ \{\mathsf{Age}\ y \mapsto y; \mathsf{Year}\ y \mapsto 2023 - y\}\\ \texttt{getAge}'\ (\Lambda\rho.\ \texttt{year}'\ (\mathsf{Age} : \mathsf{Int};\rho))\end{array}$$

The type application $x\ \cdot$ instantiates ρ with the empty closed row type $\cdot$. The above function application is well-typed because we ignore the order of labels when comparing rows ($\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}; \rho \equiv \mathsf{Year} : \mathsf{Int}; \mathsf{Age} : \mathsf{Int}; \rho$) as usual. This is the essence of the local type-only translation $\lambda^{\leqslant}_{[]} \longrightarrow \lambda^{\rho}_{[]}$ in Section 4.2.

We are relying on higher-rank polymorphism here in order to simulate upcasting on demand. For instance, an upcast on the parameter of a function of type $(\forall\rho.[\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}; \rho]) \longrightarrow B$ is simulated by instantiating ρ appropriately. We will show in Section 2.4 that restricting the target language to rank-1 polymorphism requires certain constraints on the source language.

2.2 Simple Record Subtyping as Presence Polymorphism

Now, we consider record types, through the following function.

$$\texttt{getName} = \lambda x^{\langle \mathsf{Name}:\mathsf{String}\rangle}.\ (x.\mathsf{Name})$$

The record type $\langle \mathsf{Name} : \mathsf{String}\rangle$ denotes the type of records with a single field Name containing a string. We cannot directly apply `getName` to the following record

$$\texttt{alice} = \langle \mathsf{Name} = \text{"Alice"}; \mathsf{Age} = 9\rangle$$

as the types of `alice` and x do not match. With simple record subtyping ($\lambda^{\leqslant}_{\langle\rangle}$), we can upcast $\texttt{alice} : \langle \mathsf{Name} : \mathsf{String}; \mathsf{Age} : \mathsf{Int}\rangle$ to the supertype $\langle \mathsf{Name} : \mathsf{String}\rangle$. It is intuitive to treat a record with more fields (Name and Age) as a record with fewer fields (only Name in this case).

$$\texttt{getName}\ (\texttt{alice} \rhd \langle \mathsf{Name} : \mathsf{String}\rangle)$$

Similarly to variant subtyping, we can reuse `getName` on records of different subtypes.

$$\begin{array}{l}\texttt{bob} = \langle \mathsf{Name} = \text{"Bob"}; \mathsf{Year} = 1984\rangle\\ \texttt{getName}\ (\texttt{bob} \rhd \langle \mathsf{Name} : \mathsf{String}\rangle)\end{array}$$

In a language without subtyping ($\lambda_{\langle\rangle}$), we can first deconstruct the record by projection and then reconstruct it with only the required fields, similarly to the generalised η-expansion of records.

$$\texttt{getName}\ \langle \mathsf{Name} = \texttt{alice}.\mathsf{Name}\rangle$$

[1] We omit the kinds of row variables for simplicity. They can be easily reconstructed from the contexts.

This is the essence of the local term-involved translation $\lambda^{\leqslant}_{\langle\rangle} \twoheadrightarrow \lambda_{\langle\rangle}$ in Section 4.3. Using presence polymorphism ($\lambda^{\theta}_{\langle\rangle}$), we can simulate `alice` using a type-only translation.

$$\mathtt{alice}' = \Lambda\theta_1\theta_2.\ \langle \mathsf{Name} = \text{"Alice"}; \mathsf{Age} = 9\rangle^{\langle \mathsf{Name}^{\theta_1}:\mathsf{String};\mathsf{Age}^{\theta_2}:\mathsf{Int}\rangle}$$

The presence variables θ_1 and θ_2 can be substituted with a marker indicating that the label is either present $\bullet$ or absent $\circ$. We can instantiate θ_2 with absent $\circ$ when applying `getName` to it, ignoring the Age label. This resolves the type mismatch as the equivalence relation on row types considers only present labels ($\mathsf{Name}^{\theta} : \mathsf{String} \ \equiv\ \mathsf{Name}^{\theta} : \mathsf{String}; \mathsf{Age}^{\circ} : \mathsf{Int}$). For a general translation, we must make the parameter type of `getName` presence-polymorphic, and re-abstract over `alice`′.

$$\begin{aligned}&\mathtt{getName}' = \lambda x^{\forall\theta.\langle \mathsf{Name}^{\theta}:\mathsf{String}\rangle}.\ ((x\ \bullet).\mathsf{Name})\\&\mathtt{getName}'\ (\Lambda\theta.\ \mathtt{alice}'\ \theta\ \circ)\end{aligned}$$

This is the essence of the local type-only translation $\lambda^{\leqslant}_{\langle\rangle} \longrightarrow \lambda^{\theta}_{\langle\rangle}$ in Section 4.4. The duality between variants and records is reflected by the need for dual kinds of polymorphism, namely row and presence polymorphism, which can extend or shrink rows, respectively.

2.3 Exploiting Contravariance

We have now seen how to encode simple variant subtyping as row polymorphism and simple record subtyping as presence polymorphism. These encodings embody the intuition that row polymorphism supports extending rows and presence polymorphism supports shrinking rows. However, presence polymorphism is typically treated as an optional extra for row typing. For instance, Rémy [1994] uses row polymorphism for both record and variant types, and introduces presence polymorphism only to support record extension and default cases (which fall outside the scope of our current investigation).

This naturally raises the question of whether we can encode simple record subtyping using row polymorphism alone. More generally, given the duality between records and variants, can we swap the forms of polymorphism used by the above translations?

Though row polymorphism enables extending rows and what upcasting does on record types is to remove labels, we can simulate the same behaviour by extending record types that appear in contravariant positions in a type. The duality between row and presence polymorphism can be reconciled by way of the duality between covariant and contravariant positions. Let us revisit our `getName alice` example, which we previously encoded using polymorphism. With row polymorphism ($\lambda^{\rho}_{\langle\rangle}$), we can give the function a row polymorphic type where the row variable appears in the record type of the function parameter.

$$\mathtt{getName}_{\chi} = \Lambda\rho.\lambda x^{\langle \mathsf{Name}:\mathsf{String};\rho\rangle}.\ (x.\mathsf{Name})$$

Now in order to apply $\mathtt{getName}_{\chi}$ to `alice`, we simply instantiate ρ with ($\mathsf{Age} : \mathsf{Int}$).

$$\mathtt{getName}_{\chi}\ (\mathsf{Age} : \mathsf{Int})\ \mathtt{alice}$$

Though the above example suggests a translation which only introduces type abstractions and type applications, the idea does not extend to a general composable translation. Intuitively, the main problem is that in general we cannot know which type should be used for instantiation ($\mathsf{Age} : \mathsf{Int}$ in this case) in a compositional type-only translation, which is only allowed to use the type of `getName` and `alice` $\triangleright \langle \mathsf{Name} : \mathsf{String}\rangle$. These tell us nothing about $\mathsf{Age} : \mathsf{Int}$.

In fact a much stronger result holds. In Section 4.5, we prove that there exists no type-only encoding of simple record subtyping as row polymorphism ($\lambda^{\leqslant}_{\langle\rangle} \rightsquigarrow \lambda^{\rho}_{\langle\rangle}$), and dually for variant types with presence polymorphism ($\lambda^{\leqslant}_{[]} \rightsquigarrow \lambda^{\theta}_{[]}$).

2.4 Full Subtyping as Rank-1 Polymorphism

The kind of translation sought in Section 2.3 cannot be type-only, as it would require us to know the type used for instantiation. A natural question is whether type inference can provide the type.

In order to support decidable, sound, and complete type inference, we consider a target calculus with rank-1 polymorphism ($\lambda^{\rho 1}_{\langle\rangle}$) and Hindley-Milner type inference. Now the `getName alice` example type checks without an explicit upcast or type application. [2]

$$\begin{array}{ll} \texttt{getName} = \lambda x.\,(x.\mathsf{Name}) & : \forall\rho.\langle\mathsf{Name} : \mathsf{String}; \rho\rangle \to \mathsf{String} \\ \texttt{alice} \quad = \langle\mathsf{Name} = \texttt{"Alice"}; \mathsf{Age} = 9\rangle & : \langle\mathsf{Name} : \mathsf{String}; \mathsf{Age} : \mathsf{Int}\rangle \\ \texttt{getName}\ \texttt{alice} & : \mathsf{String} \end{array}$$

Type inference automatically infers a polymorphic type for `getName`, and instantiates the variable ρ with $\mathsf{Age} : \mathsf{Int}$. This observation hints to us that we might encode terms with explicit record upcasts in $\lambda^{\rho 1}_{\langle\rangle}$ by simply erasing all upcasts (and type annotations, given that we have type inference). The global nature of erasure implies that it also works for full subtyping ($\lambda^{\leqslant\mathrm{full}}_{\langle\rangle}$) which lifts the width subtyping of rows to any type by propagating the subtyping relation to the components of type constructors. For instance, the following function upcast using full subtyping is also translated into `getName alice`, simply by erasing the upcast.

$$(\texttt{getName} \rhd (\langle\mathsf{Name} : \mathsf{String}; \mathsf{Age} : \mathsf{Int}\rangle \to \mathsf{String}))\ \texttt{alice}$$

Thus far, the erasure translation appears to work well even for full subtyping. Does it have any limitations? Yes, we must restrict the target language to rank-1 polymorphism, which can only generalise let-bound terms. The type check would fail if we were to bind `getName` via λ-abstraction and then use it at different record types. For instance, consider the following function which concatenates two names using the $+\!\!+$ operator and is applied to `getName`.

$$(\lambda f^{\langle\mathsf{Name}:\mathsf{String}\rangle\to\mathsf{String}}.\ f\ (\texttt{alice} \rhd \langle\mathsf{Name} : \mathsf{String}\rangle) +\!\!+ f\ (\texttt{bob} \rhd \langle\mathsf{Name} : \mathsf{String}\rangle))\ \texttt{getName}$$

The erasure of it is

$$(\lambda f.\ f\ \texttt{alice} +\!\!+ f\ \texttt{bob})\ \texttt{getName}$$

which is not well-typed as f can only have a monomorphic function type, whose parameter type cannot unify with both $\langle\mathsf{Name} : \mathsf{String}; \mathsf{Age} : \mathsf{Int}\rangle$ and $\langle\mathsf{Name} : \mathsf{String}; \mathsf{Year} : \mathsf{Int}\rangle$.

In order to avoid such problems, we will define an erasure translation on a restricted subcalculus of $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle}$. The key idea is to give row-polymorphic types for record manipulation functions such as `getName`. However, the above function takes a record manipulation function of type $\langle\mathsf{Name} : \mathsf{String}\rangle \to \mathsf{String}$ as a parameter, which cannot be polymorphic as we only have rank-1 polymorphism. Inspired by the notion of rank-n polymorphism, we say that a type has *rank-n records*, if no path from the root of the type (seen as an abstract syntax tree) to a record type passes to the left of n or more arrows. We define the translation only on the subcalculus $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle 2}$ of $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle}$ in which all types have rank-2 records.

Such an erasure translation underlies the local type-only translation $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle 2} \longrightarrow \lambda^{\rho 1}_{\langle\rangle}$.

We obtain a similar result for presence polymorphism. With presence polymorphism, we can make all records presence-polymorphic (similar to the translation in Section 2.2), instead of making all record manipulation functions row-polymorphic. For instance, we can infer the following types

[2] Actually, the principal type of `getName` should be $\forall\alpha\,\rho.\langle\mathsf{Name} : \alpha; \rho\rangle \to \alpha$. We ignore value type variables for simplicity.

for the `getName alice` example.

$$\begin{array}{ll}
\texttt{getName} = \lambda x.\,(x.\mathsf{Name}) & : \langle \mathsf{Name} : \mathsf{String}\rangle \to \mathsf{String} \\
\texttt{alice} \quad = \langle \mathsf{Name} = \texttt{"Alice"}; \mathsf{Age} = 9\rangle & : \forall \theta_1 \theta_2.\langle \mathsf{Name}^{\theta_1} : \mathsf{String}; \mathsf{Age}^{\theta_2} : \mathsf{Int}\rangle \\
\texttt{getName alice} & : \mathsf{String}
\end{array}$$

Consequently, records should appear only in positions that can be generalised with rank-1 polymorphism, which can be ensured by restricting $\lambda^{\leqslant\text{full}}_{\langle\rangle}$ to the subcalculus $\lambda^{\leqslant\text{full}}_{\langle\rangle 1}$ in which all types have rank-1 records. We give a local type-only translation: $\lambda^{\leqslant\text{full}}_{\langle\rangle 1} \longrightarrow \lambda^{\theta 1}_{\langle\rangle}$.

For variants, we can also define the notion of *rank-n variants* similarly. Dually to records, we can either make all variants be row-polymorphic (similar to the translation in Section 2.1) and require types to have rank-1 variants ($\lambda^{\leqslant\text{full}}_{[]1}$), or make all variant manipulation functions be presence-polymorphic and require types to have rank-2 variants ($\lambda^{\leqslant\text{full}}_{[]2}$). For instance, we can make the `getAge` function presence-polymorphic.

$$\begin{array}{l}
\texttt{getAge} = \lambda x.\,\mathbf{case}\ x\ \{\mathsf{Age}\ y \mapsto y; \mathsf{Year}\ y \mapsto 2023 - y\} : \forall \theta_1 \theta_2.[\mathsf{Age}^{\theta_1} : \mathsf{Int}; \mathsf{Year}^{\theta_2} : \mathsf{Int}] \to \mathsf{Int} \\
\texttt{year} \quad = \mathsf{Year}\ 1984 : [\mathsf{Age} : \mathsf{Int}] \\
\texttt{getAge year}
\end{array}$$

We give two type-only translations for full variant subtyping: $\lambda^{\leqslant\text{full}}_{[]1} \longrightarrow \lambda^{\rho 1}_{[]}$ and $\lambda^{\leqslant\text{full}}_{[]2} \longrightarrow \lambda^{\theta 1}_{[]}$.

We give a detailed discussion of the four erasure translations for rank-1 polymorphism with type inference in Section 6.

2.5 Strictly Covariant Record Subtyping as Presence Polymorphism

The encodings of full subtyping discussed in Section 2.4 impose restrictions on types in the source language and rely heavily on type-inference. We now consider to what extent we can support a richer form of subtyping than simple subtyping if we return our attention to target calculi with higher-rank polymorphism and no type inference.

One complication of extending simple subtyping to full subtyping is that if we permit propagation through contravariant positions, then the subtyping order is reversed. To avoid this scenario, we first consider *strictly covariant subtyping* relation derived by only propagating simple subtyping through strictly covariant positions (i.e. never to the left of any arrow). For example, the upcast $\texttt{getName} \rhd (\langle \mathsf{Name} : \mathsf{String}; \mathsf{Age} : \mathsf{Int}\rangle \to \mathsf{String})$ in Section 2.4 is ruled out. We write $\lambda^{\leqslant\text{co}}_{\langle\rangle}$ for our calculus with strictly covariant record subtyping.

Consider the function `getChildName` returning the name of the child of a person.

$$\texttt{getChildName} = \lambda x^{\langle \mathsf{Child}:\langle \mathsf{Name}:\mathsf{String}\rangle\rangle}.\ \texttt{getName}\ (x.\mathsf{Child})$$

We can apply `getChildName` to `carol` who has a daughter `alice` with the strictly covariant subtyping relation $\langle \mathsf{Name} : \mathsf{String}; \mathsf{Child} : \langle \mathsf{Name}:\mathsf{String}; \mathsf{Age}:\mathsf{Int}\rangle\rangle \leqslant \langle \mathsf{Child} : \langle \mathsf{Name}:\mathsf{String}\rangle\rangle$.

$$\begin{array}{l}
\texttt{carol} = \langle \mathsf{Name} = \texttt{"Carol"}; \mathsf{Child} = \texttt{alice}\rangle \\
\texttt{getChildName}\ (\texttt{carol} \rhd \langle \mathsf{Child} : \langle \mathsf{Name} : \mathsf{String}\rangle\rangle)
\end{array}$$

If we work in a language without subtyping ($\lambda_{\langle\rangle}$), we can still use η-expansions instead, by nested deconstruction and reconstruction.

$$\texttt{getChildName}\ \langle \mathsf{Child} = \langle \mathsf{Name} = \texttt{carol}.\mathsf{Child}.\mathsf{Name}\rangle\rangle$$

In general, we can simulate the full subtyping (not only strictly covariant subtyping) of both records and variants using this technique. The nested de- and re-construction can be reformulated into coercion functions to be more compositional [Breazu-Tannen et al. 1991]. In Section 5.1, we show the standard local term-involved translation $\lambda^{\leqslant\text{full}}_{[]\langle\rangle} \twoheadrightarrow \lambda_{[]\langle\rangle}$ formalising this idea.

However, for type-only encodings, the idea of making every record presence-polymorphic in Section 2.2 does not work directly. Following that idea, we would translate `carol` to

$$\texttt{carol}_{\chi} = \Lambda\theta'_1\theta'_2.\ \langle \ldots ; \mathsf{Child} = \texttt{alice}' \rangle^{\langle \mathsf{Name}^{\theta'_1}:\mathsf{String};\mathsf{Child}^{\theta'_2}:\forall\theta_1\theta_2.\langle \mathsf{Name}^{\theta_1}:\mathsf{String};\mathsf{Age}^{\theta_2}:\mathsf{Int}\rangle\rangle}$$

Then, as θ_1 and θ_2 are abstracted inside a record, we cannot directly instantiate θ_2 with $\circ$ to remove the Age label without deconstructing the outer record. However, we can tweak the translation by moving the quantifiers $\forall\theta_1\theta_2$ to the top-level through introducing new type abstraction and type application, which gives rise to a translation that is type-only but global.

$$\texttt{carol}' = \Lambda\theta_1\theta_2\theta_3\theta_4.\langle \ldots ; \mathsf{Child} = \texttt{alice}'\ \theta_3\ \theta_4 \rangle^{\langle \mathsf{Name}^{\theta_1}:\mathsf{String};\mathsf{Child}^{\theta_2}:\langle \mathsf{Name}^{\theta_3}:\mathsf{String};\mathsf{Age}^{\theta_4}:\mathsf{Int}\rangle\rangle}$$

Now we can remove the Name of `carol′` and Age of `alice′` by instantiating θ_1 and θ_4 with $\circ$. As for simple subtyping, we make the parameter type of `getChildName` polymorphic, and re-abstract over `carol′`.

$$\texttt{getChildName}' = \lambda x^{\forall\theta_1\theta_2.\langle \mathsf{Child}^{\theta_1}:\langle \mathsf{Name}^{\theta_2}:\mathsf{String}\rangle\rangle}.\ \texttt{getName}\ ((x\ \bullet\ \bullet).\mathsf{Child})$$
$$\texttt{getChildName}'\ (\Lambda\theta_1\theta_2.\ \texttt{carol}'\ \circ\ \theta_1\ \theta_2\ \circ)$$

This is the essence of the global type-only translation $\lambda^{\leqslant\mathrm{co}}_{\langle\rangle} \longmapsto \lambda^{\theta}_{\langle\rangle}$ in Section 5.2.

2.6 No Type-Only Encoding of Strictly Covariant Variant Subtyping as Polymorphism

We now consider whether we could exploit hoisting of quantifiers in order to encode strictly covariant subtyping for variants ($\lambda^{\leqslant\mathrm{co}}_{[\,]}$) using row polymorphism. Interestingly, we will see that this cannot work, thus breaking the symmetry between the results for records and variants we have seen so far. To understand why, consider the following example involving nested variants.

$$\texttt{data} = (\mathsf{Raw}\ \texttt{year})^{[\mathsf{Raw}:[\mathsf{Year}:\mathsf{Int}]]}$$
$$\texttt{data} \rhd [\mathsf{Raw} : [\mathsf{Year} : \mathsf{Int}; \mathsf{Age} : \mathsf{Int}]]$$

Following the idea of moving quantifiers, we can translate `data` to use a polymorphic variant, and the upcast can then be simulated by instantiation and re-abstraction.

$$\texttt{data}_{\chi} = \Lambda\rho_1\rho_2.\ (\mathsf{Raw}\ (\texttt{year}'\ \rho_2))^{[\mathsf{Raw}:[\mathsf{Year}:\mathsf{Int};\rho_2];\rho_1]}$$
$$\Lambda\rho_1\rho_2.\ \texttt{data}_{\chi}\ \rho_1\ (\mathsf{Age} : \mathsf{Int}; \rho_2)$$

So far, the translation appears to have worked. However, it breaks down when we consider the case split on a nested variant. For instance, consider the following function.

$$\texttt{parseAge} = \lambda x^{[\mathsf{Raw}:[\mathsf{Year}:\mathsf{Int}]]}.\ \mathbf{case}\ x\ \{\mathsf{Raw}\ y \mapsto \texttt{getAge}\ (y \rhd [\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}])\}$$
$$\texttt{parseAge data}$$

It uses an upcast and the `getAge` function from Section 2.1 in the case clause. We can directly pass the nested variant `data` to it.

The difficulty with encoding `parseAge` with row polymorphism is that the abstraction of the row variable for the inner record of $\texttt{data}_{\chi}$ is hoisted up to the top-level, but case split requires a monomorphic value. Thus, we must instantiate ρ_2 with Age : Int *before* performing the case split.

$$\texttt{parseAge}_{\chi} = \lambda x^{\forall\rho_1\rho_2.[\mathsf{Raw}:[\mathsf{Year}:\mathsf{Int};\rho_2];\rho_1]}.\ \mathbf{case}\ (x\ \cdot\ (\mathsf{Age} : \mathsf{Int}))\ \{\mathsf{Raw}\ y \mapsto \texttt{getAge}\ y\}$$
$$\texttt{parseAge}_{\chi}\ \texttt{data}_{\chi}$$

However, this would not yield a compositional type-only translation, as the translation of the **case** construct only has access to the types of x and the whole case clause, which provide no information about Age : Int. Moreover, even if the translation could somehow access this type information, the translation would still fail if there were multiple incompatible upcasts of y in the case clause.

$$\mathbf{case}\ x\ \{\mathsf{Raw}\ y \mapsto \ldots y \rhd [\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}] \ldots y \rhd [\mathsf{Age} : \mathsf{String}; \mathsf{Year} : \mathsf{Int}]\}$$

The first upcast requires ρ_2 to be instantiated with $\mathsf{Age} : \mathsf{Int}$ but the second requires it to be instantiated with the incompatible $\mathsf{Age} : \mathsf{String}$. The situation is no better if we add presence polymorphism. In Section 5.3, we prove that there exists no type-only encoding of strictly covariant variant subtyping as row and presence polymorphism ($\lambda_{[]}^{\leqslant \mathrm{co}} \not\rightsquigarrow \lambda_{[]}^{\rho\theta}$).

2.7 No Type-Only Encoding of Full Record Subtyping as Polymorphism

For variants, we have just seen that a type-only encoding of full subtyping does not exist, even if we restrict propagation of simple subtyping to strictly covariant positions. For records, we have seen how to encode strictly covariant subtyping with presence polymorphism by hoisting quantifiers to the top-level. We now consider whether we could somehow lift the strictly covariance restriction and encode full record subtyping with polymorphism?

The idea of hoisting quantifiers does not work arbitrarily, exactly because we cannot hoist quantifiers through contravariant positions. Moreover, presence polymorphism alone cannot extend rows. Consider the full subtyping example $\mathtt{getName} \triangleright (\langle \mathsf{Name} : \mathsf{String}; \mathsf{Age} : \mathsf{Int}\rangle \to \mathsf{String})$ from Section 2.4. The `getName` function is translated to the `getName'` function in Section 2.2, which provides no way to extend the parameter record type with $\mathsf{Age} : \mathsf{Int}$.

$$\mathtt{getName'} = \lambda x^{\forall \theta.\langle \mathsf{Name}^{\theta}:\mathsf{String}\rangle}.\ ((x\ \bullet).\mathsf{Name})$$

A tempting idea is to add row polymorphism:

$$\mathtt{getName'}_{\boldsymbol{\chi}} = \Lambda \rho.\lambda x^{\forall \theta.\langle \mathsf{Name}^{\theta}:\mathsf{String};\rho\rangle}.\ ((x\ \bullet).\mathsf{Name})$$

Now we can instantiate ρ with $\mathsf{Age} : \mathsf{Int}$ to simulate the upcast. However, this still does not work. One issue is that we have no way to remove the labels introduced by the row variable ρ in the function body, as x is only polymorphic in θ. For instance, consider the following upcast of the function `getUnit` which replaces the function body of `getName` with an upcast of x.

$$\mathtt{getUnit} = \lambda x^{\langle \mathsf{Name}:\mathsf{String}\rangle}.(x \triangleright \langle\rangle)$$
$$\mathtt{getUnit} \triangleright (\langle \mathsf{Name} : \mathsf{String}; \mathsf{Age} : \mathsf{Int}\rangle \to \langle\rangle)$$

Following the above idea, `getUnit` is translated to

$$\mathtt{getUnit}_{\boldsymbol{\chi}} = \Lambda \rho.\lambda x^{\forall \theta.\langle \mathsf{Name}^{\theta}:\mathsf{String};\rho\rangle}.x\ \circ$$

Then, in the translation of the upcast of `getUnit`, the row variable ρ is expected to be instantiated with a row containing $\mathsf{Age} : \mathsf{Int}$. However, we cannot remove $\mathsf{Age} : \mathsf{Int}$ again in the translation of the function body, meaning that the upcast inside `getUnit` cannot yield an empty record.

Section 5.4 expands on the discussion here and proves that there exists no type-only translation of unrestricted full record subtyping as row and presence polymorphism ($\lambda_{\langle\rangle}^{\leqslant \mathrm{full}} \not\rightsquigarrow \lambda_{\langle\rangle}^{\rho\theta}$).

3 CALCULI

The foundation for our exploration of relative expressive power of subtyping and parametric polymorphism is Church's simply-typed λ-calculus [Church 1940]. We extend it with variants and records, respectively. We further extend the variant calculus twice: first with simple structural subtyping and then with row polymorphism. Similarly, we also extend the record calculus twice: first with structural subtyping and then with presence polymorphism. In Section 5 and 6, we explore further extensions with strictly covariant subtyping, full subtyping and rank-1 polymorphism.

3.1 A Simply-Typed Base Calculus λ

Our base calculus is a Church-style simply typed λ-calculus, which we denote λ. Figure 2 shows the syntax, static semantics, and dynamic semantics of it. The calculus features one kind (Type)

Syntax

$$
\begin{array}{rl}
\mathsf{Kind} \ni K ::= & \mathsf{Type} \mid \underbrace{\mathsf{Row}_{\mathcal{L}}}_{[\,]\langle\rangle} \\
\mathsf{Type} \ni A, B ::= & \alpha \mid A \to B \\
\mid & \underbrace{[R]}_{[\,]} \mid \underbrace{\langle R\rangle}_{\langle\rangle} \\
\mathsf{TyEnv} \ni \Delta ::= & \cdot \mid \Delta, \alpha \\
\mathsf{Env} \ni \Gamma ::= & \cdot \mid \Gamma, x : A
\end{array}
\qquad
\begin{array}{rl}
\mathsf{Row} \ni R ::= & \underbrace{\cdot \mid \ell : A; R}_{[\,]\langle\rangle} \\
\mathsf{Term} \ni M, N ::= & x \mid \lambda x^A.M \mid M\,N \\
\mid & \underbrace{(\ell\ M)^A \mid \mathbf{case}\ M\ \{\ell_i\ x_i \mapsto N_i\}_i}_{[\,]} \\
\mid & \underbrace{\langle \ell_i = M_i\rangle_i \mid M.\ell}_{\langle\rangle} \\
\underbrace{\mathsf{Label} \supseteq \mathcal{L} \ni \ell}_{[\,]\langle\rangle} &
\end{array}
$$

Static Semantics

$\boxed{\Delta \vdash A : K}$

$$
\text{K-Base}\ \frac{}{\Delta, \alpha \vdash \alpha : \mathsf{Type}}
\qquad
\text{K-Arrow}\ \frac{\Delta \vdash A : \mathsf{Type} \qquad \Delta \vdash B : \mathsf{Type}}{\Delta \vdash A \to B : \mathsf{Type}}
\qquad
\underbrace{\text{K-EmptyRow}\ \frac{}{\Delta \vdash \cdot : \mathsf{Row}_{\mathcal{L}}}}_{[\,]\langle\rangle}
$$

$$
\underbrace{\text{K-ExtendRow}\ \frac{\Delta \vdash A : \mathsf{Type} \qquad \Delta \vdash R : \mathsf{Row}_{\mathcal{L} \uplus \{\ell\}}}{\Delta \vdash \ell : A; R : \mathsf{Row}_{\mathcal{L}}}}_{[\,]\langle\rangle}
\qquad
\underbrace{\text{K-Variant}\ \frac{\Delta \vdash R : \mathsf{Row}_{\emptyset}}{\Delta \vdash [R] : \mathsf{Type}}}_{[\,]}
\qquad
\underbrace{\text{K-Record}\ \frac{\Delta \vdash R : \mathsf{Row}_{\emptyset}}{\Delta \vdash \langle R\rangle : \mathsf{Type}}}_{\langle\rangle}
$$

$\boxed{\Delta; \Gamma \vdash M : A}$

$$
\text{T-Var}\ \frac{}{\Delta; \Gamma, x : A \vdash x : A}
\qquad
\text{T-Lam}\ \frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x^A.M : A \to B}
\qquad
\text{T-App}\ \frac{\Delta; \Gamma \vdash M : A \to B \qquad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M\,N : B}
\qquad
\underbrace{\text{T-Inject}\ \frac{(\ell : A) \in R \qquad \Delta; \Gamma \vdash M : A}{\Delta; \Gamma \vdash (\ell\ M)^{[R]} : [R]}}_{[\,]}
$$

$$
\underbrace{\text{T-Case}\ \frac{\Delta; \Gamma \vdash M : [\ell_i : A_i]_i \qquad [\Delta; \Gamma, x_i : A_i \vdash N_i : B]_i}{\Delta; \Gamma \vdash \mathbf{case}\ M\ \{\ell_i\ x_i \mapsto N_i\}_i : B}}_{[\,]}
\qquad
\underbrace{\text{T-Record}\ \frac{[\Delta; \Gamma \vdash M_i : A_i]_i}{\Delta; \Gamma \vdash \langle \ell_i = M_i\rangle_i : \langle \ell_i : A_i\rangle_i}}_{\langle\rangle}
\qquad
\underbrace{\text{T-Project}\ \frac{\Delta; \Gamma \vdash M : \langle R\rangle \qquad (\ell : A) \in R}{\Delta; \Gamma \vdash M.\ell : A}}_{\langle\rangle}
$$

Dynamic Semantics

$$
\begin{array}{ll}
\beta\text{-Lam} & (\lambda x^A.M)\ N \rightsquigarrow_\beta M[N/x] \\
\underbrace{\beta\text{-Case}}_{[\,]} & \mathbf{case}\ (\ell_j\ M)^A\ \{\ell_i\ x_i \mapsto N_i\}_i \rightsquigarrow_\beta N_j[M/x_j] \\
\underbrace{\beta\text{-Project}}_{\langle\rangle} & \langle (\ell_i = M_i)_i\rangle.\ell_j \rightsquigarrow_\beta M_j
\end{array}
$$

Fig. 2. Syntax, static semantics, and dynamic semantics of λ (unhighlighted parts), and its extensions with variants $\lambda_{[\,]}$ (highlighted parts with $[\,]$ subscript), and records $\lambda_{\langle\rangle}$ (highlighted parts with $\langle\rangle$ subscript).

to classify well-formed types. We will enrich the structure of kinds in the subsequent sections when we add rows (e.g. Sections 3.2 and 3.5). The syntactic category of types includes abstract base types (α) and the function types ($A \to B$), which classify functions with domain A and codomain B. The terms consist of variables (x), λ-abstraction ($\lambda x^A.M$) binding variable x of type A in term M, and application ($M\,N$) of M to N. We track base types in a type environment (Δ) and the type of variables in a term environment (Γ). We treat environments as unordered mappings. The static and

Syntax $\quad \text{Term} \ni M ::= \ldots \mid \boxed{M \triangleright A}_{[\,]\langle\rangle}$

Static Semantics

$\boxed{A \leqslant A'}$ $\qquad \boxed{\Delta; \Gamma \vdash M : A}$

$$\boxed{\text{S-Variant}\quad \frac{\mathrm{dom}(R) \subseteq \mathrm{dom}(R') \qquad R'|_{\mathrm{dom}(R)} = R}{[R] \leqslant [R']}}_{[\,]} \qquad \boxed{\text{T-Upcast}\quad \frac{\Delta; \Gamma \vdash M : A \qquad A \leqslant B}{\Delta; \Gamma \vdash M \triangleright B : B}}_{[\,]\langle\rangle}$$

$$\boxed{\text{S-Record}\quad \frac{\mathrm{dom}(R') \subseteq \mathrm{dom}(R) \qquad R|_{\mathrm{dom}(R')} = R'}{\langle R \rangle \leqslant \langle R' \rangle}}_{\langle\rangle}$$

Dynamic Semantics

$$\begin{array}{ll} \boxed{\triangleright\text{-Variant}}_{[\,]} & (\ell\, M)^A \triangleright B \rightsquigarrow_{\triangleright} (\ell\, M)^B \\ \boxed{\triangleright\text{-Record}}_{\langle\rangle} & \langle \ell_i = M_{\ell_i} \rangle_i \triangleright \langle \ell'_j : A_j \rangle_j \rightsquigarrow_{\triangleright} \langle \ell'_j = M_{\ell'_j} \rangle_j \end{array}$$

Fig. 3. Extensions of $\lambda_{[\,]}$ with simple subtyping $\lambda^{\leqslant}_{[\,]}$ (highlighted parts with [] subscript), and extensions of $\lambda_{\langle\rangle}$ with simple subtyping $\lambda^{\leqslant}_{\langle\rangle}$ (highlighted parts with $\langle\rangle$ subscript).

dynamic semantics are standard. We implicitly require type annotations in terms to be well-kinded, e.g., $\Delta; \Gamma \vdash \lambda x^A.M : A \to B$ requires $\Delta \vdash A$.

3.2 A Calculus with Variants $\lambda_{[\,]}$

$\lambda_{[\,]}$ is the extension of λ with variants. Figure 2 incorporates the extensions to the syntax, static semantics, and dynamic semantics. *Rows* are the basis for variants (and later records). We assume a countably infinite set of labels $\mathcal{L}_\omega$. Given a finite set of labels $\mathcal{L}$, a row of kind $\mathsf{Row}_{\mathcal{L}}$ denotes a partial mapping from the cofinite set $(\mathcal{L}_\omega \setminus \mathcal{L})$ of all labels except those in $\mathcal{L}$ to types. We say that a row of kind $\mathsf{Row}_\emptyset$ is *complete*. A variant type $([R])$ is given by a complete row R. A row is written as a sequence of pairs of labels and types. We often omit the leading $\cdot$, writing e.g. $\ell_1 : A_1, \ldots, \ell_n : A_n$ or $(\ell_i : A_i)_i$ when n is clear from context. We identify rows up to reordering of labels. Injection $(\ell\, M)^A$ introduces a term of variant type by tagging the payload M with ℓ, whose resulting type is A. A case split ($\mathbf{case}\ M\ \{\ell_i\ x_i \mapsto N_i\}_i$) eliminates an M by matching against the tags ℓ_i. A successful match on ℓ_i binds the payload of M to x_i in N_i. The kinding rules ensure that rows contain no duplicate labels. The typing rules for injections and case splits and the β-rule for variants are standard.

3.3 A Calculus with Variants and Structural Subtyping $\lambda^{\leqslant}_{[\,]}$

$\lambda^{\leqslant}_{[\,]}$ is the extension of $\lambda_{[\,]}$ with simple structural subtyping. Figure 3 shows the extensions to syntax, static semantics, and dynamic semantics.

Syntax. The explicit upcast operator $(M \triangleright A)$ coerces M to type A.

Static Semantics. The S-Variant rule asserts that variant $[R]$ is a subtype of variant $[R']$ if row R' contains at least the same label-type pairs as row R. We write $\mathrm{dom}(R)$ for the domain of row R (i.e. its labels), and $R|_{\mathcal{L}}$ for the restriction of R to the label set $\mathcal{L}$. The T-Upcast rule enables the upcast $M \triangleright B$ if the term M has type A and A is a subtype of B.

Syntax $\quad \text{Type} \ni A ::= \ldots \mid \forall \rho^K.A \qquad \text{Term} \ni M ::= \ldots \mid \Lambda \rho^K.M \mid M\,R$

$\qquad\qquad \text{Row} \ni R ::= \ldots \mid \rho \qquad \text{TyEnv} \ni \Delta ::= \ldots \mid \Delta, \rho : K$

Static Semantics

$\boxed{\Delta \vdash A : K}$ $\qquad$ $\boxed{\Delta; \Gamma \vdash M : A}$

K-RowVar
$$\frac{}{\Delta, \rho : \mathsf{Row}_{\mathcal{L}} \vdash \rho : \mathsf{Row}_{\mathcal{L}}}$$

T-RowLam
$$\frac{\Delta, \rho : K; \Gamma \vdash M : A \qquad \rho \notin \mathsf{ftv}(\Gamma)}{\Delta; \Gamma \vdash \Lambda \rho^K.M : \forall \rho^K.A}$$

K-RowAll
$$\frac{\Delta, \rho : \mathsf{Row}_{\mathcal{L}} \vdash A : \mathsf{Type}}{\Delta \vdash \forall \rho^{\mathsf{Row}_{\mathcal{L}}}.A : \mathsf{Type}}$$

T-RowApp
$$\frac{\Delta; \Gamma \vdash M : \forall \rho^K.B \qquad \Delta \vdash A : K}{\Delta; \Gamma \vdash M\,A : B[A/\rho]}$$

Dynamic Semantics

$$\tau\text{-RowLam} \qquad (\Lambda \rho^K.M)\,R \rightsquigarrow_\tau M[R/\rho]$$

Fig. 4. Extensions of $\lambda_{[\,]}$ with row polymorphism $\lambda^{\rho}_{[\,]}$.

Dynamic Semantics. The $\triangleright$-Variant reduction rule coerces an injection $(\ell\, M)$ of type A to a larger (variant) type B. We distinguish upcast rules from β rules writing instead $\rightsquigarrow_\triangleright$ for the reduction relation. Correspondingly, we write $\twoheadrightarrow_\triangleright$ for the compatible closure of $\rightsquigarrow_\triangleright$.

3.4 A Calculus with Row Polymorphic Variants $\lambda^{\rho}_{[\,]}$

$\lambda^{\rho}_{[\,]}$ is the extension of $\lambda_{[\,]}$ with row polymorphism. Figure 4 shows the extensions to the syntax, static semantics, and dynamic semantics.

Syntax. The syntax of types is extended with a quantified type $(\forall \rho^K.A)$ which binds the row variable ρ with kind K in the type A (the kinding rules restrict K to always be of kind $\mathsf{Row}_{\mathcal{L}}$ for some $\mathcal{L}$). The syntax of rows is updated to allow a row to end in a row variable (ρ). A row variable enables the tail of a row to be extended with further labels. A row with a row variable is said to be *open*; a row without a row variables is said to be closed.

Terms are extended with type (row) abstraction $(\Lambda \rho^K.M)$ binding the row variable ρ with kind K in M and row application $(M\,R)$ of M to R. Finally, type environments are updated to track the kinds of row variables.

Static Semantics. The kinding and typing rules for row polymorphism are the standard rules for System F specialised to rows.

Dynamic Semantics. The new rule τ-RowLam is the standard β rule for System F, but specialised to rows. Though it is a β rule, we use the notation $\rightsquigarrow_\tau$ to distinguish it from other β rules as it only influences types. This distinction helps us to make the meta theory of translations in Section 4 clearer. We write $\twoheadrightarrow_\tau$ for the compatible closure of $\rightsquigarrow_\tau$.

3.5 A Calculus with Records $\lambda_{\langle\rangle}$

$\lambda_{\langle\rangle}$ is λ extended with records. Figure 2 incorporates the extensions to the syntax, static semantics, and dynamic semantics. As with $\lambda_{[\,]}$, we use rows as the basis of record types. The extensions of kinds, rows and labels are the same as $\lambda_{[\,]}$. As with variants a record type $(\langle R \rangle)$ is given by a complete row R. Records introduction $\langle \ell_i = M_i \rangle_i$ gives a record in which field i has label ℓ_i and payload M_i. Record projection $(M.\ell)$ yields the payload of the field with label ℓ from the record M. The static and dynamic semantics for records are standard.

Syntax

$$
\begin{array}{rl}
\mathsf{Kind} \ni K ::= \ldots \mid \mathsf{Pre} & \mathsf{Presence} \ni P ::= \circ \mid \bullet \mid \theta \\
\mathsf{Type} \ni A ::= \ldots \mid \forall \theta . A & \mathsf{Term} \ni M ::= \ldots \mid \Lambda\theta.M \mid M\,P \mid \boxed{\langle \ell_i = M_i \rangle_i^A} \\
\mathsf{Row} \ni R ::= \ldots \mid \boxed{\ell^P : A; R} & \mathsf{TyEnv} \ni \Delta ::= \ldots \mid \Delta, \theta
\end{array}
$$

Static Semantics

$\boxed{\Delta \vdash A : K}$

$$
\text{K-Absent}\ \frac{}{\Delta \vdash \circ : \mathsf{Pre}}
\qquad
\text{K-Present}\ \frac{}{\Delta \vdash \bullet : \mathsf{Pre}}
\qquad
\text{K-PreVar}\ \frac{}{\Delta, \theta \vdash \theta : \mathsf{Pre}}
\qquad
\text{K-PreAll}\ \frac{\Delta, \theta \vdash A : \mathsf{Type}}{\Delta \vdash \forall \theta . A : \mathsf{Type}}
$$

$$
\boxed{\text{K-ExtendRow}\ \frac{\Delta \vdash P : \mathsf{Pre} \quad \Delta \vdash A : \mathsf{Type} \quad \Delta \vdash R : \mathsf{Row}_{\mathcal{L} \uplus \{\ell\}}}{\Delta \vdash \ell^P : A; R : \mathsf{Row}_{\mathcal{L}}}}
$$

$\boxed{\Delta; \Gamma \vdash M : A}$

$$
\text{T-PreLam}\ \frac{\Delta, \theta; \Gamma \vdash M : A \qquad \theta \notin \mathsf{ftv}(\Gamma)}{\Delta; \Gamma \vdash \Lambda \theta . M : \forall \theta . A}
\qquad
\text{T-PreApp}\ \frac{\Delta; \Gamma \vdash M : \forall \theta . A \qquad \Delta \vdash P : \mathsf{Pre}}{\Delta; \Gamma \vdash M\,P : A[P/\theta]}
$$

$$
\boxed{\text{T-Record}\ \frac{[\Delta; \Gamma \vdash M_i : A_i]_i}{\Delta; \Gamma \vdash \langle \ell_i = M_i \rangle_i^{\langle \ell_i^{P_i} : A_i \rangle_i} : \langle \ell_i^{P_i} : A_i \rangle_i}
\qquad
\text{T-Project}\ \frac{\Delta; \Gamma \vdash M : \langle R \rangle \qquad (\ell^\bullet : A) \in R}{\Delta; \Gamma \vdash M.\ell : A}}
$$

Dynamic Semantics

$$
\begin{array}{ll}
\boxed{\beta\text{-Project}} & \langle (\ell_i = M_i)_i \rangle^A . \ell_j \leadsto_\beta M_j \\
\tau\text{-PreLam} & (\Lambda \theta . M)\, P \leadsto_\tau M[P/\theta]
\end{array}
$$

Fig. 5. Extensions and modifications to $\lambda_{\langle\rangle}$ with presence polymorphism $\lambda_{\langle\rangle}^{\theta}$. Highlighted parts replace the old ones in $\lambda_{\langle\rangle}$, rather than extensions.

3.6 A Calculus with Records and Structural Subtyping $\lambda_{\langle\rangle}^{\leqslant}$

$\lambda_{\langle\rangle}^{\leqslant}$ is the extension of $\lambda_{\langle\rangle}$ with structural subtyping. Figure 3 shows the extensions to syntax, static semantics, and dynamic semantics. The only difference from $\lambda_{[]}^{\leqslant}$ is the subtyping rule S-Record and dynamic semantics rule $\rhd$-Record. The subtyping relation ($\leqslant$) is just like that for $\lambda_{[]}^{\leqslant}$ except R and R' are swapped. The S-Record rule states that a record type $\langle R \rangle$ is a subtype of $\langle R' \rangle$ if the row R contains at least the same label-type pairs as R'. The $\rhd$-Record rule upcasts a record $\langle \ell_i = M_i \rangle_i$ to type $\langle R \rangle$ by directly constructing a record with only the fields required by the supertype $\langle R \rangle$. We implicitly assume that the two indexes j range over the same set of integers.

3.7 A Calculus with Presence Polymorphic Records $\lambda_{\langle\rangle}^{\theta}$

$\lambda_{\langle\rangle}^{\theta}$ is the extension of $\lambda_{\langle\rangle}$ with presence-polymorphic records. Figure 5 shows the extensions to the syntax, static semantics, and dynamic semantics.

Syntax. The syntax of kinds is extended with the kind of presence types (Pre). The structure of rows is updated with presence annotations on labels $(\ell_i^{P_i} : A_i)_i$. Following Rémy [1994], a label can be marked as either absent ($\circ$), present ($\bullet$), or polymorphic in its presence (θ). Note that in either case, the label is associated with a type. Thus, it is perfectly possible to say that some label ℓ is absent with some type A. As for row variables, the syntax of types is extended with a quantified type ($\forall\theta.A$), and the syntax of terms is extended with presence abstraction ($\Lambda\theta.M$) and application ($M\ P$). To have a deterministic static semantics, we need to extend record constructions with type annotations to indicate the presence types of labels ($\langle \ell_i = M_i \rangle^A$). Finally, the structure of type environments is updated to track presence variables. With presence types, we not only ignore the order of labels, but also ignore absent labels when comparing row types. We also ignore absent labels when comparing two typed records in $\lambda^{\theta}_{\langle\rangle}$. For instance, the row $\langle \ell_1 = M; \ell_2 = N \rangle^{\langle \ell_1^{\bullet}:A; \ell_2^{\circ}:B \rangle}$ is equivalent to $\langle \ell_1 = M \rangle^{\langle \ell_1^{\bullet}:A \rangle}$.

Static Semantics. The kinding and typing rules for polymorphism (K-PreAll, T-PreLam, T-PreApp) are the standard ones for System F specialised to presence types. The first three new kinding rules K-Absent, K-Present, and K-PreVar handle presence types directly. They assign kind Pre to absent, present, and polymorphic presence annotation respectively. The kinding rule K-ExtendRow is extended with a new kinding judgement to check P is a presence type. The typing rules for records, T-Record, and projections, T-Project, are updated to accommodate the presence annotations on labels. The typing rule for record introduction, T-Record, is changed such that the type of each component coincides with the annotation. The projection rule, T-Project, is changed such that the ℓ component must be present in the record row.

Dynamic Semantics. The new rewrite rule τ-PreLam is the standard β rule for System F, but specialised to presence types. As with $\lambda^{\rho}_{[]}$ we use the notation $\rightsquigarrow_\tau$ to distinguish it from other β rules and write $\leadsto_\tau$ for its compatible closure. The β-Project* rule is the same as β-Project, but with a type annotation on the record.

4 SIMPLE SUBTYPING AS POLYMORPHISM

In this section, we consider encodings of simple subtyping. We present four encodings and two non-existence results as depicted in Fig. 1. Specifically, in addition to the standard term-involved encodings of simple variant and record subtyping in Section 4.1 and Section 4.3, we give type-only encodings of simple variant subtyping as row polymorphism in Section 4.2, and simple record subtyping as presence polymorphism in Section 4.4. For each translation, we establish its correctness by demonstrating the preservation of typing derivations and the correspondence between the operational semantics. In Section 4.5, we show the non-existence of type-only encodings if we swap the row and presence polymorphism of the target languages.

Compositional Translations. We restrict our attention to compositional translations defined inductively over the structure of derivations. For convenience we will often write these as if they are defined on plain terms, but formally the domain is derivations rather than terms, whilst the codomain is terms. In this section translations on derivations will always be defined on top of corresponding compositional translations on types, kind environments, and type environments, in such a way that we obtain a type preservation property for each translation. In Sections 5 and 6 we will allow non-compositional translations on types (as they will necessarily need to be constructed in a non-compositional global fashion, e.g., by way of a type inference algorithm).

4.1 Local Term-Involved Encoding of $\lambda^{\leqslant}_{[]}$ in $\lambda_{[]}$

We give a local term-involved compositional translation from $\lambda^{\leqslant}_{[]}$ to $\lambda_{[]}$, formalising the idea of simulating $\mathsf{age} \rhd [\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}]$ with case split and injection in Section 2.1.

$$\begin{gathered}\llbracket - \rrbracket \ : \ \mathsf{Derivation} \longrightarrow \mathsf{Term} \\ \llbracket M^{[\ell_i : A_i]_i} \rhd [R] \rrbracket = \mathbf{case}\ \llbracket M \rrbracket\ \{\ell_i\ x_i \mapsto (\ell_i\ x_i)^{[R]}\}_i\end{gathered}$$

The translation has a similar structure to the η-expansion of variants:

$$\eta\text{-Case} \qquad M^{[\ell_i:A_i]_i} \rightsquigarrow_\eta \textbf{case}\ M\ \{\ell_i\ x_i \mapsto (\ell_i\ x_i)^{[\ell_i:A_i]_i}\}_i$$

The following theorem states that the translation preserves typing derivations. Note that compositional translations always translate environments pointwise. For type environments, we have $[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : [\![A]\!]$. For kind environments, we have the identity function $[\![\Delta]\!] = \Delta$.

Theorem 4.1 (Type Preservation). *Every well-typed $\lambda^{\leqslant}_{[\,]}$ term $\Delta; \Gamma \vdash M : A$ is translated to a well-typed $\lambda_{[\,]}$ term $[\![\Delta]\!]; [\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]$.*

In order to state an operational correspondence result, we first define $\rightsquigarrow_{\beta\triangleright}$ as the union of $\rightsquigarrow_\beta$ and $\rightsquigarrow_\triangleright$, and $\leadsto_{\beta\triangleright}$ as its compatible closure. There is a one-to-one correspondence between reduction in $\lambda^{\leqslant}_{[\,]}$ and reduction in $\lambda_{[\,]}$.

Theorem 4.2 (Operational Correspondence). *For the translation $[\![-]\!]$ from $\lambda^{\leqslant}_{[\,]}$ to $\lambda_{[\,]}$, we have*

Simulation *If $M \leadsto_{\beta\triangleright} N$, then $[\![M]\!] \leadsto_\beta [\![N]\!]$.*
Reflection *If $[\![M]\!] \leadsto_\beta [\![N]\!]$, then $M \leadsto_{\beta\triangleright} N$.*

Intuitively, every step of β-reduction in $\lambda^{\leqslant}_{[\,]}$ is mapped to itself in $\lambda_{[\,]}$. For every step of upcast reduction of $M^{[R']} \triangleright [R]$ in $\lambda^{\leqslant}_{[\,]}$, the $\triangleright$-Variant rule guarantees that M must be a variant value. Thus, it is mapped to one step of β-reduction which reduces the η-expansion of M. The full proofs of type preservation and operational correspondence can be found in Appendix B.1.

4.2 Local Type-Only Encoding of $\lambda^{\leqslant}_{[\,]}$ in $\lambda^{\rho}_{[\,]}$

We give a local type-only translation from $\lambda^{\leqslant}_{[\,]}$ to $\lambda^{\rho}_{[\,]}$ by making variants row-polymorphic, as demonstrated by `year′` and `getAge′` in Section 2.1.

$$\begin{aligned} [\![-]\!] &: \mathsf{Type} \to \mathsf{Type} \\ [\![[R]]\!] &= \forall \rho^{\mathsf{Row}_R}.[[\![R]\!];\rho] \\ [\![-]\!] &: \mathsf{Row} \to \mathsf{Row} \\ [\![(\ell_i : A_i)_i]\!] &= (\ell_i : [\![A_i]\!])_i \end{aligned} \qquad \begin{aligned} [\![-]\!] &: \mathsf{Derivation} \to \mathsf{Term} \\ [\![(\ell\ M)^{[R]}]\!] &= \Lambda\rho^{\mathsf{Row}_R}.(\ell\ [\![M]\!])^{[[\![R]\!];\rho]} \\ [\![\textbf{case}\ M\ \{\ell_i\ x_i \mapsto N_i\}_i]\!] &= \textbf{case}\ ([\![M]\!]\ \cdot)\ \{\ell_i\ x_i \mapsto [\![N_i]\!]\}_i \\ [\![M^{[R]} \triangleright [R']]\!] &= \Lambda\rho^{\mathsf{Row}_{R'}}.[\![M]\!]\ @\ ([\![R'\backslash R]\!];\rho) \end{aligned}$$

The Row_R is short for $\mathsf{Row}_{\mathsf{dom}(R)}$ and $R\backslash R'$ is defined as row difference:

$$R\backslash R' = (\ell : A)_{(\ell:A)\in R \text{ and } (\ell:A)\notin R'}$$

The translation preserves typing derivations.

Theorem 4.3 (Type Preservation). *Every well-typed $\lambda^{\leqslant}_{[\,]}$ term $\Delta; \Gamma \vdash M : A$ is translated to a well-typed $\lambda^{\rho}_{[\,]}$ term $[\![\Delta]\!]; [\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]$.*

In order to state an operational correspondence result, we introduce two auxiliary reduction relations. First, we annotate the type application introduced by the translation of upcasts with the symbol @ to distinguish it from the type application introduced by the translation of **case**. We write $\rightsquigarrow_\nu$ for the associated reduction and $\leadsto_\nu$ for its compatible closure.

$$\nu\text{-RowLam} \qquad (\Lambda\rho^K.M)\ @\ A \rightsquigarrow_\nu M[A/\rho]$$

Then, we add another intuitive reduction rule for upcast in $\lambda^{\leqslant}_{[\,]}$, which allows nested upcasts to reduce to a single upcast.

$$\blacktriangleright\text{-Nested} \qquad M \triangleright A \triangleright B \rightsquigarrow_\blacktriangleright M \triangleright B$$

We write $\rightsquigarrow_{\triangleright\blacktriangleright}$ for the union of $\rightsquigarrow_\triangleright$ and $\rightsquigarrow_\blacktriangleright$, and $\leadsto_{\triangleright\blacktriangleright}$ for its compatible closure. There are one-to-one correspondences between β-reductions (modulo $\leadsto_\tau$), and between upcast and $\leadsto_\nu$.

Theorem 4.4 (Operational Correspondence). *For the translation* $[\![-]\!]$ *from* $\lambda^{\leqslant}_{[\,]}$ *to* $\lambda^{\rho}_{[\,]}$*, we have*

Simulation *If* $M \rightsquigarrow_{\beta} N$*, then* $[\![M]\!] \rightsquigarrow^{?}_{\tau}\rightsquigarrow_{\beta} [\![N]\!]$*; if* $M \rightsquigarrow_{\rhd} N$*, then* $[\![M]\!] \rightsquigarrow_{\nu} [\![N]\!]$.

Reflection *If* $[\![M]\!] \rightsquigarrow^{?}_{\tau}\rightsquigarrow_{\beta} [\![N]\!]$*, then* $M \rightsquigarrow_{\beta} N$*; if* $[\![M]\!] \rightsquigarrow_{\nu} [\![N]\!]$*, then* $M \rightsquigarrow_{\rhd\blacktriangleright} N$.

We write $\rightsquigarrow^{?}_{\tau}$ to represent zero or one step of $\rightsquigarrow_{\tau}$. For the β-reduction of a case-split in $\lambda^{\leqslant}_{[\,]}$, in order to reduce further in $\lambda^{\rho}_{[\,]}$, the translation of it must first reduce the empty row type application $[\![M]\!]\ \cdot$ by $\rightsquigarrow_{\tau}$. One step of upcast reduction in $\lambda^{\leqslant}_{[\,]}$ is simply mapped to the corresponding type application in $\lambda^{\rho}_{[\,]}$. The other direction (reflection) is slightly more involved as one step of $\rightsquigarrow_{\nu}$ in $\lambda^{\rho}_{[\,]}$ may correspond to a nested upcast; hence the need for $\rightsquigarrow_{\rhd\blacktriangleright}$ instead of $\rightsquigarrow_{\rhd}$. The proofs of type preservation and operational correspondence can be found in Appendix B.2.

4.3 Local Term-Involved Encoding of $\lambda^{\leqslant}_{\langle\rangle}$ in $\lambda_{\langle\rangle}$

We give a local term-involved translation from $\lambda^{\leqslant}_{\langle\rangle}$ to $\lambda_{\langle\rangle}$, formalising the idea of simulating `alice` $\rhd \langle \mathsf{Name} : \mathsf{String}\rangle$ with projection and record construction in Section 2.1.

$$[\![-]\!] \ : \ \mathsf{Derivation} \longrightarrow \mathsf{Term}$$
$$[\![M \rhd \langle \ell_i : A_i\rangle_i]\!] = \langle \ell_i = [\![M]\!].\ell_i\rangle_i$$

The translation has a similar structure to the η-expanding of records, which is

$$\eta\text{-Project} \qquad M^{\langle \ell_i : A_i\rangle_i} \leadsto_{\eta} \langle \ell_i = M.\ell_i\rangle_i$$

The translation preserves typing derivations.

Theorem 4.5 (Type Preservation). *Every well-typed* $\lambda^{\leqslant}_{\langle\rangle}$ *term* $\Delta;\Gamma \vdash M : A$ *is translated to a well-typed* $\lambda_{\langle\rangle}$ *term* $[\![\Delta]\!];[\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]$.

One upcast or β-reduction in $\lambda^{\leqslant}_{\langle\rangle}$ corresponds to a sequence of β-reductions in $\lambda_{\langle\rangle}$.

Theorem 4.6 (Operational Correspondence). *For the translation* $[\![-]\!]$ *from* $\lambda^{\leqslant}_{\langle\rangle}$ *to* $\lambda_{\langle\rangle}$*, we have*

Simulation *If* $M \rightsquigarrow_{\beta\rhd} N$*, then* $[\![M]\!] \rightsquigarrow^{*}_{\beta} [\![N]\!]$.

Reflection *If* $[\![M]\!] \rightsquigarrow_{\beta} N'$*, then there exists* N *such that* $N' \rightsquigarrow^{*}_{\beta} [\![N]\!]$ *and* $M \rightsquigarrow_{\beta\rhd} N$.

We write $\rightsquigarrow^{*}_{\beta}$ to represent multiple (including zero) steps of $\rightsquigarrow_{\beta}$. Unlike Theorem 4.2, one step of reduction in $\lambda^{\leqslant}_{\langle\rangle}$ might be mapped to multiple steps of reduction in $\lambda_{\langle\rangle}$ because the translation of upcast possibly introduces multiple copies of the same term. For instance, $[\![M \rhd \langle \ell_1 : A; \ell_2 : B\rangle]\!] = \langle \ell_1 = [\![M]\!].\ell_1; \ell_2 = [\![M]\!].\ell_2\rangle$. One step of β-reduction in M in $\lambda^{\leqslant}_{\langle\rangle}$ is mapped to at least two steps of β-reduction in the two copies of $[\![M]\!]$ in $\lambda_{\langle\rangle}$. Reflection is basically the reverse of simulation but requires at least one step of reduction in $\lambda_{\langle\rangle}$. The proofs of type preservation and operational correspondence can be found in Appendix B.3.

4.4 Local Type-Only Encoding of $\lambda^{\leqslant}_{\langle\rangle}$ in $\lambda^{\theta}_{\langle\rangle}$

Before presenting the translation, let us focus on order of labels in types. Though generally we treat row types as unordered collections, in this section we assume, without loss of generality, that there is a canonical order on labels, and the labels of any rows (including records) conform to this order. This assumption is crucial in preserving the correspondence between labels and presence variables bound by abstraction. For example, consider the type $A = \langle \ell_1 : A_1; \dots; \ell_n : A_n\rangle$ in $\lambda^{\leqslant}_{\langle\rangle}$. Following the idea of making records presence polymorphic as exemplified by `getName`′ and `alice`′ in Section 2.2, this record is translated as $[\![A]\!] = \forall\theta_1 \dots \theta_n.\langle \ell_1^{\theta_1} : [\![A_1]\!]; \dots; \ell_n^{\theta_n} : [\![A_n]\!]\rangle$. With the canonical order, we can guarantee that ℓ_i always appears at the i-th position in the record and possesses the presence variable bound at the i-th position. The full translation is as follows.

$$\begin{array}{rl} [\![-]\!] & : \mathsf{Type} \to \mathsf{Type} \\ [\![\langle \ell_i : A_i \rangle_i]\!] & = (\forall \theta_i)_i . \langle \ell_i^{\theta_i} : [\![A_i]\!] \rangle_i \end{array}$$

$$\begin{array}{rl} [\![-]\!] & : \mathsf{Derivation} \to \mathsf{Term} \\ [\![\langle \ell_i = M_i \rangle_i^{\langle \ell_i : A_i \rangle_i}]\!] & = (\Lambda \theta_i)_i . \langle \ell_i = [\![M_i]\!] \rangle_i^{\langle \ell_i^{\theta_i} : [\![A_i]\!] \rangle_i} \\ [\![M^{\langle \ell_i : A_i \rangle_i} . \ell_j]\!] & = ([\![M]\!]\ (P_i)_i) . \ell_j \\ & \quad \text{where } P_i = \circ\,, i \neq j \quad P_j = \bullet \\ [\![M^{\langle \ell_i : A_i \rangle_i} \rhd \langle \ell'_j : A'_j \rangle_j]\!] & = (\Lambda \theta_j)_j . [\![M]\!]\ (@\, P_i)_i \\ & \quad \text{where } P_i = \circ\,, \ell_i \notin (\ell'_j)_j \quad P_i = \theta_j\,, \ell_i = \ell'_j \end{array}$$

The translation preserves typing derivations.

Theorem 4.7 (Type Preservation). *Every well-typed $\lambda^{\leqslant}_{\langle\rangle}$ term $\Delta;\Gamma \vdash M : A$ is translated to a well-typed $\lambda^{\theta}_{\langle\rangle}$ term $[\![\Delta]\!];[\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]$.*

Similarly to Section 4.2, we annotate type applications introduced by the translation of upcast with @, and write $\leadsto_\nu$ for the associated reduction rule and $\rightsquigarrow_\nu$ for its compatible closure.

$$\nu\text{-PreLam} \qquad (\Lambda\theta.M)\ @\ P \leadsto_\nu M[P/\theta]$$

We also re-use the ►-Nested reduction rule defined in Section 4.2. There is a one-to-one correspondence between β-reductions (modulo $\rightsquigarrow_\tau$), and a correspondence between one upcast reduction and a sequence of $\rightsquigarrow_\nu$ reductions.

Theorem 4.8 (Operational Correspondence). *The translation $[\![-]\!]$ from $\lambda^{\leqslant}_{\langle\rangle}$ to $\lambda^{\theta}_{\langle\rangle}$ has the following properties:*

Simulation *If $M \rightsquigarrow_\beta N$, then $[\![M]\!] \rightsquigarrow_\tau^* \rightsquigarrow_\beta [\![N]\!]$; if $M \rightsquigarrow_\rhd N$, then $[\![M]\!] \rightsquigarrow_\nu^* [\![N]\!]$.*

Reflection *If $[\![M]\!] \rightsquigarrow_\tau^* \rightsquigarrow_\beta [\![N]\!]$, then $M \rightsquigarrow_\beta N$; if $[\![M]\!] \rightsquigarrow_\nu N'$, then there exists N such that $N' \rightsquigarrow_\nu^* [\![N]\!]$ and $M \rightsquigarrow_{\rhd\blacktriangleright} N$.*

Unlike Theorem 4.4, one step of reduction in $\lambda^{\leqslant}_{\langle\rangle}$ might be mapped to multiple steps of reduction in $\lambda^{\theta}_{\langle\rangle}$ because we might need to reduce the type application of multiple presence types in the translation results of projection and upcast. Reflection is again basically the reverse of simulation, requiring at least one step of reduction in $\lambda^{\theta}_{\langle\rangle}$. The proofs of type preservation and operational correspondence can be found in Appendix B.4.

4.5 Swapping Row and Presence Polymorphism

In Section 4.2 and Section 4.4, we encode simple subtyping for variants using row polymorphism, and simple subtyping for records using presence polymorphism. These encodings enjoy the property that they only introduce new type abstractions and applications. A natural question is whether we can swap the polymorphism used by the encodings meanwhile preserve the type-only property. As we have seen in Section 2.3, an intuitive attempt to encode simple record subtyping with row polymorphism failed. Specifically, we have the problematic translation

$$\begin{array}{rl} & [\![\mathtt{getName}\ (\mathtt{alice} \rhd \langle \mathsf{Name} : \mathsf{String} \rangle)]\!] \\ = & [\![\mathtt{getName}]\!]\ (\mathsf{Age} : \mathsf{Int})\ [\![\mathtt{alice} \rhd \langle \mathsf{Name} : \mathsf{String} \rangle]\!] \\ = & \mathtt{getName}_{\boldsymbol{\chi}}\ (\mathsf{Age} : \mathsf{Int})\ \mathtt{alice} \end{array}$$

First, the type information Age : Int is not accessible to a compositional type-only translation of the function application here. Moreover, the type preservation property is also broken: $[\![\mathtt{alice} \rhd \langle \mathsf{Name} : \mathsf{String} \rangle]\!]$ should have type $[\![\langle \mathsf{Name} : \mathsf{String} \rangle]\!]$, but here it is just translated to `alice` itself, which has an extra label Age in its record type. We give a general non-existence theorem.

Theorem 4.9. *There exists no global type-only encoding of $\lambda^{\leqslant}_{\langle\rangle}$ in $\lambda^{\rho}_{\langle\rangle}$, and no global type-only encoding of $\lambda^{\leqslant}_{[\,]}$ in $\lambda^{\theta}_{[\,]}$.*

The extensions for $\lambda^{\rho}_{\langle\rangle}$ and $\lambda^{\theta}_{[]}$ are straightforward and can be found in Appendix A. The proofs of this theorem can be found in Appendix E.1. We will give further non-existence results in Section 5. The core idea underlying the proofs of this kind of non-existence result is to construct counterexamples and use proof by contradiction. One important observation is that in our case a type-only translation ensures that terms are invariant under the translation modulo type abstraction and type application. As a consequence, we may characterise the general form of any such translation by accounting for the possibility of adding type abstractions and type applications in every possible position. Then we can obtain a contradiction by considering the general form of type-only translations of carefully selected terms.

To give an example, let us consider the proof of Theorem 4.9. Consider $\langle\rangle$ and $\langle \ell = y\rangle \rhd \langle\rangle$ which have the same type under environments $\Delta = \alpha_0$ and $\Gamma = y : \alpha_0$. Any type-only translation must yield $[\![\langle\rangle]\!] = \Lambda\overline{\alpha}.\langle\rangle$ and

$$[\![\langle \ell = y\rangle \rhd \langle\rangle]\!] = \Lambda\overline{\beta}.[\![\langle \ell = y\rangle]\!]\ \overline{B} = \Lambda\overline{\beta}.(\Lambda\overline{\alpha}'.\langle \ell = [\![y]\!]\ \overline{A}'\rangle)\ \overline{B} = \Lambda\overline{\beta}.(\Lambda\overline{\alpha}'.\langle \ell = (\Lambda\overline{\beta}'.y)\ \overline{A}'\rangle)\ \overline{B}$$

which can be simplified to $\Lambda\overline{\gamma}.\langle \ell = \Lambda\overline{\delta}.y\rangle$. Thus, $[\![\langle\rangle]\!]$ has type $\forall\overline{\alpha}.\langle\rangle$, and $[\![\langle \ell = y\rangle \rhd \langle\rangle]\!]$ has type $\forall\overline{\gamma}.\langle \ell : \forall\overline{\delta}.\alpha_0\rangle$. By type preservation, they should still have the same type, which implies $\forall\overline{\alpha}.\langle\rangle = \forall\overline{\gamma}.\langle \ell : \forall\overline{\delta}.\alpha_0\rangle$. However, this equation obviously does not hold, showing a contradiction.

The above proof relies on the assumption that translations should always satisfy the type preservation theorem. Sometimes this assumption can be too strong. In order to show the robustness of our theorem, we provide three proofs of Theorem 4.9 in Appendix E.1, where only one of them relies on type preservation. The second proof uses the compositionality and a similar argument to the `getName`$_{\boldsymbol{\chi}}$ example in Section 2.3, while the third proof does not rely on either of them.

In Section 6, we will show that it is possible to simulate record subtyping with rank-1 row polymorphism and type inference, at the cost of a weaker type preservation property and some extra conditions on the source language.

5 FULL SUBTYPING AS POLYMORPHISM

So far we have only considered simple subtyping, which means the subtyping judgement applies shallowly to a single variant or record constructor (width subtyping). Any notion of simple subtyping can be mechanically lifted to full subtyping by inductively propagating the subtyping relation to the components of each type. The direction of the subtyping relation remains the same for covariant positions, and is reversed for contravariant positions.

In this section, we consider encodings of full subtyping. We first formalise the calculus $\lambda^{\leqslant\text{full}}_{[]\langle\rangle}$ with full subtyping for records and variants, and give its standard term-involved translation to $\lambda_{[]\langle\rangle}$ (Section 5.1). Next we give a type-only encoding of strictly covariant record subtyping (Section 5.2) and a non-existence result for variants (Section 5.3). Finally, we give a non-existence result for type-only encodings of full record subtyping as polymorphism (Section 5.4).

5.1 Local Term-Involved Encoding of $\lambda^{\leqslant\text{full}}_{[]\langle\rangle}$ in $\lambda_{[]\langle\rangle}$

We first consider encoding $\lambda^{\leqslant\text{full}}_{[]\langle\rangle}$, an extension of $\lambda^{\leqslant}_{[]}$ and $\lambda^{\leqslant}_{\langle\rangle}$ with full subtyping, in $\lambda_{[]\langle\rangle}$, the combination of $\lambda_{[]}$ and $\lambda_{\langle\rangle}$. Figure 6 shows the standard full subtyping rules of $\lambda^{\leqslant\text{full}}_{[]\langle\rangle}$. We inductively propagate the subtyping relation to sub-types, and reverse the subtyping order for function parameters because of contravariance. The reflexivity and transitivity rules are admissible.

For the dynamic semantics of $\lambda^{\leqslant\text{full}}_{[]\langle\rangle}$, one option is to give concrete upcast rules for each value constructor, similar to $\lambda^{\leqslant}_{[]}$ and $\lambda^{\leqslant}_{\langle\rangle}$. However, as encoding full subtyping is more intricate than encoding simple subtyping (especially the encoding in Section 5.2), upcast reduction rules significantly complicate the operational correspondence theorems. To avoid such complications we adopt

$\boxed{A \leqslant A'}$

$$\text{FS-Var}\ \frac{}{\alpha \leqslant \alpha} \qquad \text{FS-Fun}\ \frac{A' \leqslant A \qquad B \leqslant B'}{A \to B \leqslant A' \to B'}$$

$$\text{FS-Variant}\ \frac{\mathrm{dom}(R) \subseteq \mathrm{dom}(R') \quad [A_i \leqslant A'_i]_{(\ell_i : A_i) \in R, (\ell_i : A'_i) \in R'}}{[R] \leqslant [R']} \qquad \text{FS-Record}\ \frac{\mathrm{dom}(R') \subseteq \mathrm{dom}(R) \quad [A_i \leqslant A'_i]_{(\ell_i : A_i) \in R, (\ell_i : A'_i) \in R'}}{\langle R \rangle \leqslant \langle R' \rangle}$$

Fig. 6. Full subtyping rules of $\lambda^{\leqslant \text{full}}_{[\,]\langle\rangle}$.

an *erasure semantics* for $\lambda^{\leqslant \text{full}}_{[\,]\langle\rangle}$ which, following Pierce [2002], interprets upcasts as no-ops. The type erasure function $\mathrm{erase}(-)$ transforms typed terms in $\lambda^{\leqslant \text{full}}_{[\,]\langle\rangle}$ to untyped terms in $\lambda_{[\,]\langle\rangle}$ by erasing all upcasts and type annotations. It is given by the homomorphic extension of the following equations.

$$\mathrm{erase}(M \rhd A) = \mathrm{erase}(M) \quad \mathrm{erase}(\lambda x^A.M) = \lambda x.\mathrm{erase}(M) \quad \mathrm{erase}((\ell\, M)^A) = \ell\, \mathrm{erase}(M)$$

We show a correspondence between the upcast rules and the erasure semantics in Appendix C.2. In the following, we always use the erasure semantics for calculi with full subtyping or strictly covariant subtyping.

The idea of the local term-involved translation from $\lambda^{\leqslant \text{full}}_{[\,]\langle\rangle}$ to $\lambda_{[\,]\langle\rangle}$ in Section 2.5 has been well-studied as the *coercion semantics* of subtyping [Breazu-Tannen et al. 1991, 1990; Pierce 2002], which transforms subtyping relations $A \leqslant B$ into coercion functions $[\![A \leqslant B]\!]$. Writing translations in the form of coercion functions ensures compositionality. The translation is standard and shown in Appendix C.1. For instance, the full subtyping relation in Section 2.5 is translated to

$$\begin{aligned}
& [\![\langle \mathsf{Name} : \mathsf{String}; \mathsf{Child} : \langle \mathsf{Name} : \mathsf{String}; \mathsf{Age} : \mathsf{Int} \rangle\rangle \leqslant \langle \mathsf{Child} : \langle \mathsf{Name} : \mathsf{String} \rangle\rangle]\!] \\
= {} & (\lambda x.\langle \mathsf{Child} = [\![\langle \mathsf{Name} : \mathsf{String}; \mathsf{Age} : \mathsf{Int}\rangle \leqslant \langle \mathsf{Name} : \mathsf{String}\rangle]\!]\ x.\mathsf{Child}\rangle) \\
= {} & \lambda x.\langle \mathsf{Child} = (\lambda x.\langle \mathsf{Name} = x.\mathsf{Name}\rangle)\ x.\mathsf{Child}\rangle \\
\rightsquigarrow^*_\beta {} & \lambda x.\langle \mathsf{Child} = \langle \mathsf{Name} = x.\mathsf{Child}.\mathsf{Name}\rangle\rangle
\end{aligned}$$

We refer the reader to Pierce [2002] and Breazu-Tannen et al. [1990] for the standard type preservation and operational correspondence theorems and proofs.

5.2 Global Type-Only Encoding of $\lambda^{\leqslant \text{co}}_{\langle\rangle}$ in $\lambda^{\theta}_{\langle\rangle}$

As a stepping stone towards exploring the possibility of type-only encodings of full subtyping, we first consider an easier problem: the encoding of $\lambda^{\leqslant \text{co}}_{\langle\rangle}$, a calculus with strictly covariant structural subtyping for records. Strictly covariant subtyping lifts simple subtyping through only the covariant positions of all type constructors. For $\lambda^{\leqslant \text{co}}_{[\,]\langle\rangle}$, the only change with respect to $\lambda^{\leqslant \text{full}}_{[\,]\langle\rangle}$ is to replace the subtyping rule FS-Fun with the following rule which requires the parameter types to be equal:

$$\frac{B \leqslant B'}{A \to B \leqslant A \to B'}$$

As illustrated by the examples $\texttt{carol}_{\chi}$ and $\texttt{carol}'$ from Section 2.5, we can extend the idea of encoding simple record subtyping as presence polymorphism described in Section 4.4 by hoisting quantifiers to the top-level, yielding a global but type-only encoding of $\lambda^{\leqslant \text{co}}_{\langle\rangle}$ in $\lambda^{\theta}_{\langle\rangle}$. The full type and term translations are spelled out in Figure 7 together with three auxiliary functions.

As in Section 4.4, we rely on a canonical order on labels. The auxiliary function $[\![A, \overline{P}]\!]$ instantiates a polymorphic type A with $\overline{P}$, simulating the type application in the term level. The auxiliary function $(\!|\theta, A|\!)$ takes a presence variable θ and a type A, and generates a sequence of presence variables based on θ that have the same length as the presence variables bound by $[\![A]\!]$. It is used to allocate

$$
\begin{aligned}
\llbracket - \rrbracket &: \mathsf{Type} \to \mathsf{Type} \\
\llbracket A \to B \rrbracket &= \forall \overline{\theta}. \llbracket A \rrbracket \to \llbracket B, \overline{\theta} \rrbracket \\
&\qquad \text{where } \overline{\theta} = \llparenthesis \theta, B \rrparenthesis \\
\llbracket \langle \ell_i : A_i \rangle_i \rrbracket &= \forall (\theta_i)_i\, (\overline{\theta}_i)_i . \langle \ell_i^{\theta_i} : \llbracket A_i, \overline{\theta}_i \rrbracket \rangle_i \\
&\qquad \text{where } \overline{\theta}_i = \llparenthesis \theta_i, A_i \rrparenthesis
\end{aligned}
$$

$$
\begin{aligned}
\llbracket -, - \rrbracket &: (\mathsf{Type}, \overline{\mathsf{Pre}}) \to \mathsf{Type} \\
\llbracket A, \overline{P} \rrbracket &= A' [\overline{P} / \overline{\theta}'] \\
&\qquad \text{where } \forall \overline{\theta}'. A' = \llbracket A \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
\llparenthesis -, - \rrparenthesis &: (\mathsf{Pre}, \mathsf{Type}) \to \overline{\mathsf{Pre}} \\
\llparenthesis P, \alpha \rrparenthesis &= \cdot \\
\llparenthesis P, A \to B \rrparenthesis &= \llparenthesis P, B \rrparenthesis \\
\llparenthesis P, \langle \ell_i : A_i \rangle_i \rrparenthesis &= (P_i)_i\, \llparenthesis P_i, A_i \rrparenthesis_i \\
&\qquad \text{where } P_i = \theta_i \text{ , } P \text{ is a variable } \theta \\
&\qquad\qquad P_i = \circ \text{ , } P = \circ \\
&\qquad\qquad P_i = \bullet \text{ , } P = \bullet
\end{aligned}
$$

$$
\begin{aligned}
\llbracket - \rrbracket &: \mathsf{Derivation} \to \mathsf{Term} \\
\llbracket \lambda x^A . M^B \rrbracket &= \Lambda \overline{\theta}. \lambda x^{\llbracket A \rrbracket}. \llbracket M \rrbracket\ \overline{\theta} \\
&\qquad \text{where } \overline{\theta} = \llparenthesis \theta, B \rrparenthesis \\
\llbracket M^A\ N^B \rrbracket &= \Lambda \overline{\theta}. (\llbracket M \rrbracket\ \overline{\theta})\ \llbracket N \rrbracket \\
&\qquad \text{where } \overline{\theta} = \llparenthesis \theta, A \rrparenthesis \\
\llbracket \langle \ell_i = M_i^{A_i} \rangle_i \rrbracket &= \Lambda (\theta_i)_i\, (\overline{\theta}_i)_i . \langle \ell_i = \llbracket M_i \rrbracket\ \overline{\theta}_i \rangle_i^{\langle \ell_i^{\theta_i} : \llbracket A_i \rrbracket \rangle_i} \\
&\qquad \text{where } \overline{\theta}_i = \llparenthesis \theta_i, A_i \rrparenthesis \\
\llbracket M^{\langle \ell_i : A_i \rangle_i} . \ell_j \rrbracket &= \Lambda \overline{\theta}. (\llbracket M \rrbracket\ (P_i)_i\ (\overline{P}_i)_{i<j}\ \overline{\theta}\ (\overline{P}_i)_{j<i}).\ell_j \\
&\qquad \text{where } P_i = \circ \text{ , } i \neq j \qquad \overline{\theta} = \llparenthesis \theta, A_j \rrparenthesis \\
&\qquad\qquad P_j = \bullet \qquad\qquad \overline{P}_i = \llparenthesis \circ, A_i \rrparenthesis \\
\llbracket M^A \rhd B \rrbracket &= \Lambda \overline{\theta}. \llbracket M \rrbracket\ \overline{P} \\
&\qquad \text{where } (\overline{\theta}, \overline{P}) = \llparenthesis \theta, A \leqslant B \rrparenthesis
\end{aligned}
$$

$$
\begin{aligned}
\llparenthesis -, - \rrparenthesis &: (\mathsf{Pre}, \mathsf{Type} \leqslant \mathsf{Type}) \to (\overline{\mathsf{Pre}}, \overline{\mathsf{Pre}}) \\
\llparenthesis \theta, \alpha \leqslant \alpha \rrparenthesis &= (\cdot, \cdot) \\
\llparenthesis \theta, A \to B \leqslant A \to B' \rrparenthesis &= \llparenthesis \theta, B \leqslant B' \rrparenthesis \\
\llparenthesis \theta, \langle \ell_i : A_i \rangle_i \leqslant \langle \ell'_j : A'_j \rangle_j \rrparenthesis &= ((\theta_j)_j\ (\overline{\theta}_j)_j, (P_i)_i\ (\overline{P}_i)_i) \\
&\text{where } (\overline{\theta}_j, \overline{P}'_j) = \llparenthesis \theta_j, A_i \leqslant A'_j \rrparenthesis, \ell_i = \ell'_j \\
&\qquad P_i = \circ \text{ , } \ell_i \notin (\ell'_j)_j \qquad \overline{P}_i = \llparenthesis \circ, A_i \rrparenthesis \text{ , } \ell_i \notin (\ell'_j)_j \\
&\qquad P_i = \theta_j \text{ , } \ell_i = \ell'_j \qquad \overline{P}_i = \overline{P}'_j \qquad \text{ , } \ell_i = \ell'_j
\end{aligned}
$$

Fig. 7. A global type-only translation from $\lambda_{\langle\rangle}^{\leqslant \mathrm{co}}$ to $\lambda_{\langle\rangle}^{\theta}$.

a fresh presence variable for every label in records on strictly covariant positions. We can also use it to generate a sequence of $\bullet$ or $\circ$ for the instantiation of $\llbracket A \rrbracket$ by $\llparenthesis \bullet, A \rrparenthesis$ and $\llparenthesis \circ, A \rrparenthesis$. The auxiliary function $\llparenthesis \theta, A \leqslant B \rrparenthesis$ takes a presence variable θ and a subtyping relation $A \leqslant B$, and returns a pair $(\overline{\theta}, \overline{P})$. The sequence of presence variables $\overline{\theta}$ is the same as $\llparenthesis \theta, B \rrparenthesis$. The sequence of presence types are used to instantiate $\llbracket A \rrbracket$ to get $\llbracket B \rrbracket$ (as illustrated by the term translation $\llbracket M^A \rhd B \rrbracket = \Lambda \overline{\theta}. \llbracket M \rrbracket\ \overline{P}$ which has type $\llbracket B \rrbracket$).

The translation on types is straightforward. We not only introduce a presence variable for every element of record types, but also move the quantifiers of the types of function bodies and record elements to the top level, as they are on strictly covariant positions. While the translation on terms (derivations) may appear complicated, it mainly focuses on moving type abstractions to the top level by type application and re-abstraction using the auxiliary functions. For the projection and upcast cases, it also instantiates the sub-terms with appropriate presence types. Notice that for function application $M\ N$, we only need to move the type abstractions in $\llbracket M \rrbracket$, and for projection $M.\ell_j$, we only need to move the type abstractions in the payload of ℓ_j.

Strictly speaking, the type translation is actually not compositional because of the type application introduced by the term translation. As a consequence, in the type translation, we need to use the auxiliary function $\llbracket A, \overline{P} \rrbracket$ which looks into the concrete structure of $\llbracket A \rrbracket$ instead of using it compositionally. However, we believe that it is totally fine to slightly compromise the compositionality of the type translation, which is much less interesting than the compositionality of the term translation. Moreover, we can still make the type translation compositional by extending the type syntax with type operators and type-level type application of System Fω.

We have the following type preservation theorem. The proof shown in Appendix C.3 follows from induction on typing derivations of $\lambda_{\langle\rangle}^{\leqslant \mathrm{co}}$.

Theorem 5.1 (Type Preservation). *Every well-typed $\lambda_{\langle\rangle}^{\leqslant \mathrm{co}}$ term $\Delta; \Gamma \vdash M : A$ is translated to a well-typed $\lambda_{\langle\rangle}^{\theta}$ term $\llbracket \Delta \rrbracket; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket A \rrbracket$.*

In order to state an operational correspondence result, we use the erasure semantics for $\lambda^{\theta}_{\langle\rangle}$ given by the standard type erasure function defined as the homomorphic extension of the following equations.

$$\mathsf{erase}(\Lambda\theta.M) = \mathsf{erase}(M) \quad \mathsf{erase}(M\ P) = \mathsf{erase}(M) \quad \mathsf{erase}(\lambda x^A.M) = \lambda x.\mathsf{erase}(M)$$

Since the terms in $\lambda^{\leqslant\mathrm{co}}_{\langle\rangle}$ and $\lambda^{\theta}_{\langle\rangle}$ are both erased to untyped $\lambda_{\langle\rangle}$, for the operational correspondence we need only show that any term in $\lambda^{\leqslant\mathrm{co}}_{\langle\rangle}$ is still erased to the same term after translation.

Theorem 5.2 (Operational Correspondence). *The translation* $[\![-]\!]$ *from* $\lambda^{\leqslant\mathrm{co}}_{\langle\rangle}$ *to* $\lambda^{\theta}_{\langle\rangle}$ *satisfies the equation* $\mathsf{erase}(M) = \mathsf{erase}([\![M]\!])$ *for any well-typed term* M *in* $\lambda^{\leqslant\mathrm{co}}_{\langle\rangle}$.

Proof. By straightforward induction on M. □

By using erasure semantics, the operational correspondence becomes concise and obvious for type-only translations, as all constructs introduced by type-only translations are erased by type erasure functions. It is also possible to reformulate Theorem 4.4 and Theorem 4.8 to use erasure semantics, but the current versions are somewhat more informative and not excessively complex.

5.3 Non-Existence of Type-Only Encodings of $\lambda^{\leqslant\mathrm{co}}_{[]}$ in $\lambda^{\rho\theta}_{[]}$

As illustrated by the example $\texttt{parseAge}_{\chi}\ \texttt{data}_{\chi}$ in Section 2.6, the approach of hoisting quantifiers to the top-level does not work for variants, because of case splits. Formally, we have the following general non-existence theorem showing that no other approaches exist.

Theorem 5.3. *There exists no global type-only encoding of* $\lambda^{\leqslant\mathrm{co}}_{[]}$ *in* $\lambda^{\rho\theta}_{[]}$.

The idea of the proof is the same as that of Theorem 4.9 which we have shown in Section 4.5: construct the schemes of type-only translations for certain terms and derive a contradiction. The terms we choose here are the nested variant $M = (\ell\ (\ell\ y)^{[\ell]})^{[\ell:[\ell]]}$ for some free term variable y in the environment together with its upcast $M_1 = M \triangleright [\ell : [\ell; \ell']]$ and its case split $M_2 = \textbf{case}\ M\ \{\ell\ x \mapsto x \triangleright [\ell; \ell']\}$, similar to the counterexamples we give in Section 2.6. To obtain a contradiction, we show that we cannot give a uniform type-only translation of M such that both M_1 and M_2 can be translated compositionally. The details of the proof can be found in Appendix E.2.

As a corollary, there can be no global type-only encoding of $\lambda^{\leqslant\mathrm{full}}_{[]}$ in $\lambda^{\rho\theta}_{[]}$.

One might worry that Theorem 5.3 contradicts the duality between records and variants, especially in light of Blume et al. [2006]'s translation from variants with default cases to records with record extensions. In their translation, a variant is translated to a function which takes a record of functions. For instance, the translation of variant types is:

$$[\![[\ell_i : A_i]_i]\!] = \forall\alpha.\langle \ell_i : A_i \to \alpha\rangle_i \to \alpha$$

In fact, there is no contradiction because a variant in a covariant position corresponds to a record in a contravariant position, which means that the encoding of $\lambda^{\leqslant\mathrm{co}}_{\langle\rangle}$ in Section 5.2 cannot be used. Moreover, the translation from variants to records is not type-only as it introduces λ-abstractions.

5.4 Non-Existence of Type-Only Encodings of $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle}$ in $\lambda^{\rho\theta}_{\langle\rangle}$

As illustrated by the examples $\texttt{getName'}_{\chi}$ and $\texttt{getUnit}_{\chi}$ in Section 2.7, one attempt to simulate full record subtyping by both making record types presence-polymorphic and adding row variables for records in contravariant positions fails. In fact no such encoding exists.

Theorem 5.4. *There exists no global type-only encoding of* $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle}$ *in* $\lambda^{\rho\theta}_{\langle\rangle}$.

Again, the proof idea is to give general forms of type-only translations for certain terms and proof by contradiction. Our choice of terms here are different from the counterexamples in Section 2.7 this time. Instead, we first consider two functions $f_1 = \lambda x^{\langle\rangle}.x$ and $f_2 = \lambda x^{\langle\rangle}.\langle\rangle$ of the same type $\langle\rangle \to \langle\rangle$. Any type-only translations of these functions must yield terms of the following forms:

$$\begin{aligned}
[\![f_1]\!] &= \Lambda\overline{\alpha}_1.\lambda x^{A_1}.\Lambda\overline{\beta}_1.\ \ x\ \ \overline{B}_1 \\
[\![f_2]\!] &= \Lambda\overline{\alpha}_2.\lambda x^{A_2}.\Lambda\overline{\beta}_2.[\![\langle\rangle]\!]\ \overline{B}_2 = \Lambda\overline{\alpha}_2.\lambda x^{A_2}.\Lambda\overline{\beta}_2.(\Lambda\overline{\gamma}.\langle\rangle)\ \overline{B}_2
\end{aligned}$$

By type preservation, they should have the same type, which means $x\ \overline{B}_1$ and $(\Lambda\overline{\gamma}.\langle\rangle)\ \overline{B}_2$ should also have the same type. As a result, the type A_1 of x cannot contain any type variables bound in $\overline{\alpha}_1$ unless they are inside the type of some labels which are instantiated to absent by the type application $x\ \overline{B}_1$. Then, it is problematic when we want to upcast the parameter of f_1 to be a wider record, e.g., $f_1 \rhd (\langle \ell : \langle\rangle\rangle \to \langle\rangle)$. Intuitively, because A_1 cannot be an open record type with the row variable bound in $\overline{\alpha}_1$, we actually have no way to expand A_1, which leads to a contradiction. The full proof can be found in Appendix E.3.

6 FULL SUBTYPING AS RANK-1 POLYMORPHISM

In Section 4.5, we showed that no type-only encoding of record subtyping as row polymorphism exists. The main obstacle is a lack of type information for instantiation. By focusing on rank-1 polymorphism in the target language, we need no longer concern ourselves with type abstraction and application explicitly anymore. Instead we defer to Hindley-Milner type inference [Damas and Milner 1982] as demonstrated by the examples in Section 2.4. In this section, we formalise the encodings of full subtyping as rank-1 polymorphism.

Here we focus on the encoding of $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle}$ in $\lambda^{\rho 1}_{\langle\rangle}$, a ML-style calculus with records and rank-1 row polymorphism (the same idea applies to each combination of encoding records or variants as rank-1 row polymorphism or rank-1 presence polymorphism). The specification of $\lambda^{\rho 1}_{\langle\rangle}$ is given in Appendix A.3, which uses a standard declarative Hindley-Milner style type system and extends the term syntax with let-binding $\mathbf{let}\, x = M\, \mathbf{in}\, N$ for polymorphism. We also extend $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle}$ with let-binding syntax and its standard typing and operational semantics rules.

As demonstrated in Section 2.4, we can use the following (local and type-only) erasure translation to encode $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle 2}$, the fragment of $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle}$ where types are restricted to have rank-2 records, in $\lambda^{\rho 1}_{\langle\rangle}$.

$$\begin{aligned}
[\![-]\!] &: \mathsf{Derivation} \to \mathsf{Term} \\
[\![M \rhd A]\!] &= M
\end{aligned}$$

Since the types of translated terms in $\lambda^{\rho 1}_{\langle\rangle}$ are given by type inference, we do not need to use a translation on types in the translation on terms. Moreover, we implicitly allow type annotations on λ-abstractions to be erased as they no longer exist in the target language.

To formalise the definition of rank-n records defined in Section 2.4, we introduce the predicate $\mho^n(A)$ defined as follows for any natural number n.

$$\begin{aligned}
\mho^n(\alpha) &= \mathsf{true} & \mho^0(\alpha) &= \mathsf{true} \\
\mho^n(A \to B) &= \mho^{n-1}(A) \wedge \mho^n(B) & \mho^0(A \to B) &= \mho^0(A) \wedge \mho^0(B) \\
\mho^n(\langle \ell_i : A_i\rangle_i) &= \wedge_i \mho^n(A_i) & \mho^0(\langle \ell_i : A_i\rangle_i) &= \mathsf{false}
\end{aligned}$$

We define a type A to have rank-n records, if $\mho^n(A)$ holds. The predicate $\mho^n(A)$ basically means no record types can appear in the left subtrees of n or more arrows.

The operational correspondence of the erasure translation comes for free. Note that both $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle 2}$ and $\lambda^{\rho 1}_{\langle\rangle}$ are type erased to untyped $\lambda_{\langle\rangle}$. The type erasure function of $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle 2}$ inherited from $\lambda^{\leqslant\mathrm{full}}_{[]\langle\rangle}$

in Section 5.1 is identical to the erasure translation. The type erasure function $\mathsf{erase}(-)$ of $\lambda^{\rho 1}_{\langle\rangle}$ is simply the identity function (as there is no type annotation at all). We have the following theorem.

Theorem 6.1 (Operational Correspondence). *The translation* $[\![-]\!]$ *from* $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle 2}$ *to* $\lambda^{\rho 1}_{\langle\rangle}$ *satisfies the equation* $\mathsf{erase}(M) = \mathsf{erase}([\![M]\!])$ *for any well-typed term* M *in* $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle 2}$.

Proof. By definition of $\mathsf{erase}(-)$ and $[\![-]\!]$. □

Proving type preservation is more of a challenge. To avoid the complexity of reasoning about type inference, we state the type preservation theorem using the declarative type system of $\lambda^{\rho 1}_{\langle\rangle}$, which requires us to give translations on types. We define the translations on types and environments in Figure 8. As in Section 4.4 and Section 5.2, we assume a canonical order on labels and require all rows and records to conform to this order. The translation on type environments is still the identity $[\![\Delta]\!] = \Delta$. To define the translation on term environments, we need to explicitly distinguish between variables bound by λ and variables bound by **let**. We write a, b for the former, and x, y for the latter. Because the translation on term environments may introduce fresh free type variables which are not in the original type environments, we define $[\![\Delta;\Gamma]\!]$ as a shortcut for $([\![\Delta]\!], \mathsf{ftv}([\![\Gamma]\!])); [\![\Gamma]\!]$.

The type translation $[\![A]\!]$ returns a type scheme. It uses the auxiliary translation $[\![A]\!]^*$ which extends all records types appearing strictly covariantly in A with fresh row variables, and binds all these variables at the top-level. The translation $[\![A]\!]$ opens up row types in A that appear strictly covariantly inside the left-hand-side of strictly covariant function types (by applying the auxiliary translation $[\![-]\!]^*$ to function parameter types) and binds all of the freshly generated row variables at the top-level.

We define four auxiliary functions for the translation. The functions $(\!|\rho, A|\!)$ and $(\!|\rho, A|\!)^*$ are used to generate fresh row variables. The $(\!|\rho, A|\!)$ takes a row variable ρ and a type A, and generates a sequence of row variables based on ρ with the same length of row variables bound by $[\![A]\!]$. The function $(\!|\rho, A|\!)^*$ does the same thing for $[\![A]\!]^*$. The functions $[\![A, \overline{\rho}]\!]$ and $[\![A, \overline{\rho}]\!]^*$ instantiate polymorphic types, simulating term-level type application. As we discussed in Section 5.2, these functions actually break the compositionality of the type translation, because they must inspect the concrete structure of $[\![A]\!]$. However, we only use the type translation in the theorem and proof; the compositionality of the erasure translation itself remains intact.

After giving the type and environment translation, we aim for a weak type preservation theorem which allows the translated terms to have subtypes of the original terms, because the erasure translation ignores all upcasts. As we have row variables in $\lambda^{\rho 1}_{\langle\rangle}$, the types of translated terms may contain extra row variables in strictly covariant positions. We need to define an auxiliary subtype relation $\preccurlyeq$ which only considers row variables.

$$\frac{}{\alpha \preccurlyeq \alpha} \qquad \frac{[A_i \preccurlyeq A'_i]_i}{\langle \ell_i : A_i\rangle_i \preccurlyeq \langle \ell_i : A'_i\rangle_i} \qquad \frac{[A_i \preccurlyeq A'_i]_i}{\langle (\ell_i : A_i)_i ; \rho\rangle \preccurlyeq \langle \ell_i : A'_i\rangle_i} \qquad \frac{B \preccurlyeq B'}{A \to B \preccurlyeq A \to B'} \qquad \frac{\tau \preccurlyeq \tau'}{\forall \rho^K.\tau \preccurlyeq \forall \rho^K.\tau'}$$

Finally, we have the following weak type preservation theorem.

Theorem 6.2 (Weak Type Preservation). *Every well-typed* $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle 2}$ *term* $\Delta;\Gamma \vdash M : A$ *is translated to a well-typed* $\lambda^{\rho 1}_{\langle\rangle}$ *term* $[\![\Delta;\Gamma]\!] \vdash [\![M]\!] : \tau$ *for some* $A' \leqslant A$ *and* $\tau \preccurlyeq [\![A']\!]$.

The proof makes use of $\lambda^{\leqslant\mathrm{afull}}_{\langle\rangle 2}$, an algorithmic variant of the type system of $\lambda^{\leqslant\mathrm{full}}_{\langle\rangle 2}$ which combines T-App and T-Upcast into one rule T-AppSub, and removes all explicit upcasts in terms.

$$\textsc{T-AppSub}\quad \frac{\Delta;\Gamma \vdash M : A \to B \qquad \Delta;\Gamma \vdash N : A' \qquad A' \leqslant A}{\Delta;\Gamma \vdash M\,N : B}$$

$$
\begin{aligned}
[\![-]\!] &: \mathsf{Type} \to \mathsf{TypeScheme} \\
[\![A \to B]\!] &= \forall \overline{\rho}_1 \overline{\rho}_2 . [\![A, \overline{\rho}_1]\!]^* \to [\![B, \overline{\rho}_2]\!] \\
& \quad \text{where } \overline{\rho}_1 = (\!|\rho_1, A|\!)^*,\ \overline{\rho}_2 = (\!|\rho_2, B|\!) \\
[\![\langle \ell_i : A_i \rangle_i]\!] &= \forall (\overline{\rho}_i)_i . \langle \ell_i : [\![A_i, \overline{\rho}_i]\!] \rangle_i \\
& \quad \text{where } \overline{\rho}_i = (\!|\rho_i, A_i|\!)
\end{aligned}
\qquad
\begin{aligned}
[\![-]\!]^* &: \mathsf{Type} \to \mathsf{TypeScheme} \\
[\![A \to B]\!]^* &= \forall \overline{\rho} . A \to [\![B, \overline{\rho}]\!]^* \\
& \quad \text{where } \overline{\rho} = (\!|\rho, B|\!)^* \\
[\![\langle \ell_i : A_i \rangle_i]\!]^* &= \forall \rho\, (\overline{\rho}_i)_i . \langle \ell_i : [\![A_i, \overline{\rho}_i]\!]^* ; \rho \rangle_i \\
& \quad \text{where } \overline{\rho}_i = (\!|\rho_i, A_i|\!)^*
\end{aligned}
$$

$$
\begin{aligned}
[\![-,-]\!] &: (\mathsf{Type}, \overline{\mathsf{RowVar}}) \to \mathsf{Type} \\
[\![A, \overline{\rho}]\!] &= A'[\overline{\rho}/\overline{\rho}'] \text{ where } \forall \overline{\rho}'.A' = [\![A]\!]
\end{aligned}
\qquad
\begin{aligned}
[\![-,-]\!]^* &: (\mathsf{Type}, \overline{\mathsf{RowVar}}) \to \mathsf{Type} \\
[\![A, \overline{\rho}]\!]^* &= A'[\overline{\rho}/\overline{\rho}'] \text{ where } \forall \overline{\rho}'.A' = [\![A]\!]^*
\end{aligned}
$$

$$
\begin{aligned}
(\!|-,-|\!) &: (\mathsf{RowVar}, \mathsf{Type}) \to \overline{\mathsf{RowVar}} \\
(\!|\rho, \alpha|\!) &= \cdot \\
(\!|\rho, A \to B|\!) &= (\!|\rho_1, A|\!)^*\ (\!|\rho_2, B|\!) \\
(\!|\rho, \langle \ell_i : A_i \rangle_i|\!) &= (\!|\rho_i, A_i|\!)_i
\end{aligned}
\qquad
\begin{aligned}
(\!|-,-|\!)^* &: (\mathsf{RowVar}, \mathsf{Type}) \to \overline{\mathsf{RowVar}} \\
(\!|\rho, \alpha|\!)^* &= \cdot \\
(\!|\rho, A \to B|\!)^* &= (\!|\rho, B|\!)^* \\
(\!|\rho, \langle \ell_i : A_i \rangle_i|\!)^* &= \rho\ (\!|\rho_i, A_i|\!)^*_i
\end{aligned}
$$

$$
\begin{aligned}
[\![-]\!] &: \mathsf{Env} \to \mathsf{Env} \\
[\![\cdot]\!] &= \cdot \\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : [\![A]\!] \\
[\![\Gamma, a : A]\!] &= [\![\Gamma]\!], a : [\![A, (\!|\rho_{|\Gamma|}, A|\!)^*]\!]^*
\end{aligned}
$$

Fig. 8. The translations of types and environments from $\lambda^{\leqslant\text{full}}_{\langle\rangle 2}$ to $\lambda^{\rho 1}_{\langle\rangle}$.

It is standard that $\lambda^{\leqslant\text{afull}}_{\langle\rangle 2}$ is sound and complete with respect to $\lambda^{\leqslant\text{full}}_{\langle\rangle 2}$ [Pierce 2002]. Immediately, we have that $\Delta; \Gamma \vdash M : A$ in $\lambda^{\leqslant\text{full}}_{\langle\rangle 2}$ implies $\Delta; \Gamma \vdash \widehat{M} : A'$ in $\lambda^{\leqslant\text{afull}}_{\langle\rangle 2}$ for some $A' \leqslant A$, where $\widehat{M}$ is defined as M with all upcasts erased. Thus, we only need to prove that $\Delta; \Gamma \vdash M : A$ in $\lambda^{\leqslant\text{afull}}_{\langle\rangle 2}$ implies $[\![\Delta; \Gamma]\!] \vdash [\![M]\!] : \tau$ for some $\tau \leqslant [\![A]\!]$ in $\lambda^{\rho 1}_{\langle\rangle}$. The remaining proof can be done by induction on the typing derivations in $\lambda^{\leqslant\text{afull}}_{\langle\rangle 2}$, where the most non-trivial case is the T-AppSub rule. The core idea is to use instantiation in $\lambda^{\rho 1}_{\langle\rangle}$ to simulate the subtyping relation $A' \leqslant A$ in the T-AppSub rule. This is possible because the source language $\lambda^{\leqslant\text{afull}}_{\langle\rangle 2}$ is restricted to have rank-2 records, which implies that $A \to B$ is translated to a polymorphic type where the record types in parameters are open and can be extended to simulate the subtyping relation. The full proof can be found in Appendix D.1.

So far, we have formalised the erasure translation from $\lambda^{\leqslant\text{full}}_{\langle\rangle 2}$ to $\lambda^{\rho 1}_{\langle\rangle}$. As shown in Section 2.4, we have three other results. For records, we have another erasure translation from $\lambda^{\leqslant\text{full}}_{\langle\rangle 1}$, the fragment of $\lambda^{\leqslant\text{full}}_{\langle\rangle}$ where types are restricted to have rank-1 records, to $\lambda^{\theta 1}_{\langle\rangle}$ with rank-1 presence polymorphism. Similarly, for variants, we formally define a type A to have rank-n variants, if the predicate $\Omega^n(A)$ defined as follows holds.

$$
\begin{aligned}
\Omega^n(\alpha) &= \mathsf{true} \\
\Omega^n(A \to B) &= \Omega^{n-1}(A) \wedge \Omega^n(B) \\
\Omega^n([\ell_i : A_i]_i) &= \wedge_i \Omega^n(A_i)
\end{aligned}
\qquad
\begin{aligned}
\Omega^0(\alpha) &= \mathsf{true} \\
\Omega^0(A \to B) &= \Omega^0(A) \wedge \Omega^0(B) \\
\Omega^0([\ell_i : A_i]_i) &= \mathsf{false}
\end{aligned}
$$

We also have two erasure translations from $\lambda^{\leqslant\text{full}}_{[]1}$ to $\lambda^{\rho 1}_{[]}$ and from $\lambda^{\leqslant\text{full}}_{[]2}$ to $\lambda^{\theta 1}_{[]}$. They all use the same idea that let the type inference infer row/presence-polymorphic types for terms involving records/variants, and use instantiation to automatically simulate subtyping. We omit the metatheory of these three results as they are similar to what we have seen for the encoding of $\lambda^{\leqslant\text{full}}_{\langle\rangle 2}$ in $\lambda^{\rho 1}_{\langle\rangle}$.

The requirement of rank-1 polymorphism and Hindley-Milner type inference for target languages is not mandatory; target languages can support higher-rank polymorphism via more powerful type inference algorithms like FreezeML [Emrich et al. 2020], as long as no type annotation is needed to infer rank-1 polymorphic types. One might hope to also relax the $\mho^2(-)$ restriction in $\lambda^{\leqslant\text{full}}_{\langle\rangle 2}$ by using type inference for higher-rank polymorphism. However, at least the erasure translation do not work anymore. For instance, consider the functions $\texttt{id} = \lambda x^{\langle \ell:\mathsf{Int}\rangle}.x$ and $\texttt{const} = \lambda x^{\langle \ell:\mathsf{Int}\rangle}.\langle \ell = 1\rangle$ with the same type $\langle \ell : \mathsf{Int}\rangle \to \langle \ell : \mathsf{Int}\rangle$. Type inference would give $[\![\texttt{id}]\!]$ the type $\forall \rho^{\mathsf{Row}_{\{\ell\}}}.\langle \ell : \mathsf{Int}; \rho\rangle \to \langle \ell : \mathsf{Int}; \rho\rangle$, and $[\![\texttt{const}]\!]$ the type $\forall \rho^{\mathsf{Row}_{\{\ell\}}}.\langle \ell : \mathsf{Int}; \rho\rangle \to \langle \ell : \mathsf{Int}\rangle$. For a second-order function of type $(\langle \ell : \mathsf{Int}\rangle \to \langle \ell : \mathsf{Int}\rangle) \to A$, we cannot give a type to the parameter of the function after translation which can be unified with the types of both $[\![\texttt{id}]\!]$ and $[\![\texttt{const}]\!]$. We leave it to future work to explore whether there exist other translations making use of type inference for higher-rank polymorphism.

7 DISCUSSION

We have now explored a range of encodings of structural subtyping for variants and records as parametric polymorphism under different conditions. These encodings and non-existence results capture the extent to which row and presence polymorphism can simulate structural subtyping and crystallise longstanding folklore and informal intuitions. In the remainder of this section we briefly discuss record extensions and default cases (Section 7.1), combining subtyping and polymorphism (Section 7.2), related work (Section 7.3) and conclusions and future work (Section 7.4).

7.1 Record Extensions and Default Cases

Two important extensions to row and presence polymorphism are record extensions [Rémy 1994], and its dual, default cases [Blume et al. 2006]. These operations provide extra expressiveness beyond structural subtyping. For example, with default cases, we can give a default age 42 to the function `getAge` in Section 2.1, and then apply it to variants with arbitrary constructors.

$$\begin{array}{l}\texttt{getAgeD} : \ \forall \rho^{\mathsf{Row}_{\{\mathsf{Age},\mathsf{Year}\}}}.[\mathsf{Age} : \mathsf{Int}; \mathsf{Year} : \mathsf{Int}; \rho] \to \mathsf{Int} \\ \texttt{getAgeD} = \lambda x.\ \mathbf{case}\ x\,\{\mathsf{Age}\ y \mapsto y; \mathsf{Year}\ y \mapsto 2023 - y; z \mapsto 42\} \\ \texttt{getAgeD}\ (\mathsf{Name}\ \text{"Carol"}) \rightsquigarrow^*_\beta 42\end{array}$$

7.2 Combining Subtyping and Polymorphism

Though row and presence polymorphism can simulate subtyping well and support expressive extensions like record extension and default cases, it can still be beneficial to allow both subtyping and polymorphism together in the same language. For example, the OCaml programming language combines row and presence polymorphism with subtyping. Row and presence variables are hidden in its core language. It supports both polymorphic variants and polymorphic objects (a variation on polymorphic records) as well as explicit upcast for closed variants and records. Our results give a rationalisation for why OCaml supports subtyping in addition to row polymorphism. Row polymorphism simply is not expressive enough to give a local encoding of unrestricted structural subtyping, even though OCaml indirectly supports full first-class polymorphism.

Bounded quantification [Cardelli et al. 1994; Cardelli and Wegner 1985] extends system F with subtyping by introducing subtyping bounds to type variables. There is also much work on the type inference for both polymorphism and subtyping based on collecting, solving, and simplifying constraints [Pottier 1998, 2001; Trifonov and Smith 1996]. Algebraic subtyping [Dolan 2016; Dolan and Mycroft 2017] combines subtyping and parametric polymorphism, offering compact principal types and decidable subsumption checking. MLstruct [Parreaux and Chau 2022] extends algebraic subtyping with intersection and union types, giving rise to another alternative to row polymorphism.

7.3 Related Work

Row types. Wand [1987] first introduced rows and row polymorphism. There are many further papers on row types, which take a variety of approaches, particularly focusing on extensible records.

Harper and Pierce [1990] extended System F with constrained quantification, where predicates ρ lacks L and ρ has L are used to indicate the presence and absence of labels in row variables. Gaster and Jones [1996] and Gaster [1998] explore a calculus with a similar lacks predicate based on qualified types. Rémy [1989] introduced the concept of presence types and polymorphism, and Rémy [1994] combines row and presence polymorphism. Leijen [2005] proposed a variation on row polymorphism with support for scoped labels. Pottier and Rémy [2004] consider type inference for row and presence polymorphism in HM(X). Morris and McKinna [2019] introduce Rose, an algebraic foundation for row typing via a rather general language with two predicates representing the containment and combination of rows. It is parametric over a row theory which enables it to express different styles of row types (including Wand and Rémy's style and Leijen's style).

Row polymorphism vs structural subtyping. Wand [1987] compares his calculus with row polymorphism (similar to $\lambda^{\rho 1}_{[\,]\langle\rangle}$) with Cardelli [1984]'s calculus with structural subtyping (similar to $\lambda^{\leqslant\text{full}}_{[\,]\langle\rangle}$) and shows that they cannot be encoded in each other by examples. Pottier [1998] conveys the intuition that row polymorphism can lessen the need for subtyping to some extent, but there are still situations where subtyping are necessary, e.g., the reuse of λ-bound variables which cannot be polymorphic given only rank-1 polymorphism.

Disjoint polymorphism. Disjoint intersection types [d. S. Oliveira et al. 2016] generalise record types. Record concatenation and restriction [Cardelli and Mitchell 1991] are replaced by a merge operator [Dunfield 2014] and a type difference operator [Xu et al. 2023], respectively. Parametric polymorphism of disjoint intersection types is supported via disjoint polymorphism [Alpuim et al. 2017] where type variables are associated with disjointness constraints. Similarly to our work, Xie et al. [2020] prove that both row polymorphism and bounded quantification of record types can be encoded in terms of disjoint polymorphism.

7.4 Conclusion and Future Work

We carried out a formal and systematic study of the encoding of structural subtyping as parametric polymorphism. To better reveal the relative expressive power of these two type system features, we introduced the notion of type-only translations to avoid the influence of non-trivial term reconstruction. We gave type-only translations from various calculi with subtyping to calculi with different kinds of polymorphism and proved their correctness; we also proved a series of non-existence results. Our results provide a precise characterisation of the long-standing folklore intuition that row polymorphism can often replace subtyping. Additionally, they offer insight into the trade-offs between subtyping and polymorphism in the design of programming languages.

In future we would like to explore whether it might be possible to extend our encodings relying on type inference to systems supporting higher-rank polymorphism such as FreezeML [Emrich et al. 2020]. We would also like to consider other styles of row typing such as those based on scoped labels [Leijen 2005] and Rose [Morris and McKinna 2019]. In addition to variant and record types, row types are also the foundation for various effect type systems, e.g. for effect handlers [Hillerström and Lindley 2016; Leijen 2017]. It would be interesting to investigate to what extent our approach can be applied to effect typing. Aside from studying the relationship between subtyping and row and presence polymorphism we would also like to study the ergonomics of row and presence polymorphism in practice, especially their compatibility with other programming language features such as algebraic data types.

REFERENCES

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory* *and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 1–28. https://doi.org/10.1007/978-3-662-54434-1_1

Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. 1979. Simula Begin. Studentlitteratur (Lund, Sweden), Bratt Institut fuer nues Lernen (Goch, FRG), Charwell-Bratt Ltd (Kent, England).

Matthias Blume, Umut A. Acar, and Wonseok Chae. 2006. Extensible programming with first-class cases. In *ICFP*. ACM, 239–250.

Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. 1991. Inheritance as implicit coercion. *Information and Computation* 93, 1 (1991), 172–221. https://doi.org/10.1016/0890-5401(91)90055-7 Selections from 1989 IEEE Symposium on Logic in Computer Science.

Val Breazu-Tannen, Carl A. Gunter, and Andre Scedrov. 1990. Computing with Coercions. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, Gilles Kahn (Ed.). ACM, 44–60. https://doi.org/10.1145/91556.91590

Luca Cardelli. 1984. A Semantics of Multiple Inheritance. In *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings (Lecture Notes in Computer Science, Vol. 173)*, Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin (Eds.). Springer, 51–67. https://doi.org/10.1007/3-540-13346-1_2

Luca Cardelli. 1988. Structural Subtyping and the Notion of Power Type. In *POPL*. ACM Press, 70–79.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1994. An Extension of System F with Subtyping. *Inf. Comput.* 109, 1/2 (1994), 4–56. https://doi.org/10.1006/inco.1994.1013

Luca Cardelli and John C. Mitchell. 1991. Operations on Records. *Math. Struct. Comput. Sci.* 1, 1 (1991), 3–48. https://doi.org/10.1017/S0960129500000049

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (1985), 471–522. https://doi.org/10.1145/6041.6042

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *J. Symb. Log.* 5, 2 (1940), 56–68.

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 364–377. https://doi.org/10.1145/2951913.2951945

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

Stephen Dolan. 2016. *Algebraic Subtyping*. Ph. D. Dissertation. Computer Laboratory, University of Cambridge, United Kingdom.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *POPL*. ACM, 60–72.

Jana Dunfield. 2014. Elaborating intersection and union types. *J. Funct. Program.* 24, 2-3 (2014), 133–165. https://doi.org/10.1017/S0956796813000270

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: Complete and Easy Type Inference for First-Class Polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 423–437. https://doi.org/10.1145/3385412.3386003

Matthias Felleisen. 1991. On the Expressive Power of Programming Languages. *Sci. Comput. Program.* 17, 1-3 (1991), 35–75. Revised version.

Benedict R Gaster. 1998. *Records, variants and qualified types.* Ph. D. Dissertation. University of Nottingham.

Benedict R Gaster and Mark P Jones. 1996. *A polymorphic type system for extensible records and variants.* Technical Report. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University ….

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph. D. Dissertation. Université Paris 7, France.

Robert William Harper and Benjamin C. Pierce. 1990. Extensible records without subsumption. (2 1990). https://doi.org/10.1184/R1/6605507.v1

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

Daan Leijen. 2005. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05), Tallinn, Estonia*. https://www.microsoft.com/en-us/research/publication/extensible-records-with-scoped-labels/

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 486–499. https://doi.org/10.1145/3009837.3009872

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.* 3, POPL (2019), 12:1–12:28.

Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: principal type inference in a Boolean algebra of structural types. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 449–478. https://doi.org/10.1145/3563304

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

François Pottier. 1998. *Type Inference in the Presence of Subtyping: from Theory to Practice.* Research Report RR-3483. INRIA. https://hal.inria.fr/inria-00073205

François Pottier. 2001. Simplifying Subtyping Constraints: A Theory. *Inf. Comput.* 170, 2 (2001), 153–183. https://doi.org/10.1006/inco.2001.2963

François Pottier and Didier Rémy. 2004. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, Chapter 10, 460–489. https://doi.org/10.7551/mitpress/1104.003.0016

Didier Rémy. 1989. Typechecking Records and Variants in a Natural Extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989.* ACM Press, 77–88. https://doi.org/10.1145/75277.75284

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

John C. Reynolds. 1974. Towards a theory of type structure. In *Symposium on Programming (LNCS, Vol. 19)*. Springer, 408–423.

John C. Reynolds. 1980. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation (Lecture Notes in Computer Science, Vol. 94)*. Springer, 211–258.

Valery Trifonov and Scott F. Smith. 1996. Subtyping Constrained Types. In *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1145)*, Radhia Cousot and David A. Schmidt (Eds.). Springer, 349–365. https://doi.org/10.1007/3-540-61739-6_52

Mitchell Wand. 1987. Complete Type Inference for Simple Objects. In *LICS*. IEEE Computer Society, 37–44.

Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and Bounded Polymorphism via Disjoint Polymorphism. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.27

Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2023. Making a Type Difference: Subtraction on Intersection Types as Generalized Record Operations. *Proc. ACM Program. Lang.* 7, POPL (2023), 893–920. https://doi.org/10.1145/3571224

A MORE CALCULI

In this section, we show the specifications of some calculi appearing in the paper.

A.1 A Calculus with Row Polymorphic Records $\lambda^{\rho}_{\langle\rangle}$

The extensions to the syntax, static semantics, and dynamic semantics of $\lambda_{\langle\rangle}$ for a calculus with row polymorphic records are shown in Figure 9. Actually, they are exactly the same as the extensions to $\lambda_{[\,]}$ for $\lambda^{\rho}_{[\,]}$ in Figure 4.

Syntax

$$\mathsf{Type} \ni A ::= \ldots \mid \forall \rho^{K}.A \qquad \mathsf{Term} \ni M ::= \ldots \mid \Lambda \rho^{K}.M \mid M\,R$$

$$\mathsf{Row} \ni R ::= \ldots \mid \rho \qquad \mathsf{TyEnv} \ni \Delta ::= \ldots \mid \Delta, \rho : K$$

Static Semantics

$\boxed{\Delta \vdash A : K}$ $\boxed{\Delta; \Gamma \vdash M : A}$

$$\textsf{K-RowVar}\quad \frac{}{\Delta, \rho : \mathsf{Row}_{\mathcal{L}} \vdash \rho : \mathsf{Row}_{\mathcal{L}}} \qquad \textsf{T-RowLam}\quad \frac{\Delta, \rho : K; \Gamma \vdash M : A \qquad \rho \notin \mathsf{ftv}(\Gamma)}{\Delta; \Gamma \vdash \Lambda \rho^{K}.M : \forall \rho^{K}.A}$$

$$\textsf{K-RowAll}\quad \frac{\Delta, \rho : \mathsf{Row}_{\mathcal{L}} \vdash A : \mathsf{Type}}{\Delta \vdash \forall \rho^{\mathsf{Row}_{\mathcal{L}}}.A : \mathsf{Type}} \qquad \textsf{T-RowApp}\quad \frac{\Delta; \Gamma \vdash M : \forall \rho^{K}.B \qquad \Delta \vdash A : K}{\Delta; \Gamma \vdash M\,A : B[A/\rho]}$$

Dynamic Semantics

$$\tau\textsf{-RowLam} \qquad (\Lambda \rho^{K}.M)\,R \leadsto_{\tau} M[R/\rho]$$

Fig. 9. Extensions of $\lambda_{\langle\rangle}$ with row polymorphism $\lambda^{\rho}_{\langle\rangle}$

A.2 A Calculus with Presence Polymorphic Variants $\lambda^{\theta}_{[\,]}$

The extensions and modifications to the syntax, static semantics, and dynamic semantics of $\lambda_{[\,]}$ for a calculus with presence polymorphic variants $\lambda^{\theta}_{[\,]}$ are shown in Figure 10.

One thing worth noting is that in T-Case, we do not require all labels in the type of M to be present, which is dual to the T-Record rule in Figure 5. It does not loss any generality as our equivalence relation between rows only considers present labels.

A.3 A Calculus with Rank-1 Row Polymorphic Records $\lambda^{\rho 1}_{\langle\rangle}$

The extensions to the syntax, static semantics, and dynamic semantics for $\lambda^{\rho 1}_{\langle\rangle}$, a calculus with records and rank-1 row polymorphism are shown in Figure 11. For the type syntax, we introduce row variables and type schemes. For the term syntax, we drop the type annotation on λ abstractions, and add the **let** syntax for polymorphism. We only give the declarative typing rules, as the syntax-directed typing rules and type inference are just standard [Damas and Milner 1982]. Notice that we do not introduce type variables for values in type schemes for simplicity. The lack of principal types is fine here as we are working with declarative typing rules. It is easy to regain principal types by adding value type variables.

B PROOFS OF ENCODINGS IN SECTION 4

In this section, we show the proofs of type preservation and operational correspondence for all the four translations in Section 4.

Syntax

$$\begin{array}{ll} \text{Kind} \ni K ::= \ldots \mid \mathsf{Pre} & \text{Presence} \ni P ::= \circ \mid \bullet \mid \theta \\ \text{Type} \ni A ::= \ldots \mid \forall \theta.A & \text{Term} \ni M ::= \ldots \mid \Lambda\theta.M \mid M\,P \\ \text{Row} \ni R ::= \ldots \mid \boxed{\ell^P : A; R} & \text{TyEnv} \ni \Delta ::= \ldots \mid \Delta, \theta \end{array}$$

Static Semantics

$\boxed{\Delta \vdash A : K}$

$$\text{K-Absent}\ \frac{}{\Delta \vdash \circ : \mathsf{Pre}} \qquad \text{K-Present}\ \frac{}{\Delta \vdash \bullet : \mathsf{Pre}} \qquad \text{K-PreVar}\ \frac{}{\Delta, \theta \vdash \theta : \mathsf{Pre}} \qquad \text{K-PreAll}\ \frac{\Delta, \theta \vdash A : \mathsf{Type}}{\Delta \vdash \forall\theta.A : \mathsf{Type}}$$

$$\text{K-ExtendRow}\ \frac{\Delta \vdash P : \mathsf{Pre} \qquad \Delta \vdash A : \mathsf{Type} \qquad \Delta \vdash R : \mathsf{Row}_{\mathcal{L} \uplus \{\ell\}}}{\Delta \vdash \ell^P : A; R : \mathsf{Row}_{\mathcal{L}}}$$

$\boxed{\Delta; \Gamma \vdash M : A}$

$$\text{T-PreLam}\ \frac{\Delta, \theta; \Gamma \vdash M : A \qquad \theta \notin \mathsf{ftv}(\Gamma)}{\Delta; \Gamma \vdash \Lambda\theta.M : \forall\theta.A} \qquad \text{T-PreApp}\ \frac{\Delta; \Gamma \vdash M : \forall\theta.A \qquad \Delta \vdash P : \mathsf{Pre}}{\Delta; \Gamma \vdash M\,P : A[P/\theta]}$$

$$\text{T-Inject}\ \frac{(\ell^\bullet : A) \in R \qquad \Delta; \Gamma \vdash M : A}{\Delta; \Gamma \vdash (\ell\, M)^{[R]} : [R]} \qquad \text{T-Case}\ \frac{\Delta; \Gamma \vdash M : [\ell_i^{P_i} : A_i]_i \qquad [\Delta; \Gamma, x_i : A_i \vdash N_i : B]_i}{\Delta; \Gamma \vdash \mathbf{case}\ M\ \{\ell_i\ x_i \mapsto N_i\}_i : B}$$

Dynamic Semantics

$$\tau\text{-PreLam} \qquad (\Lambda\theta.M)\ P \leadsto_\tau M[P/\theta]$$

Fig. 10. Extensions and modifications to $\lambda_{[]}$ with presence polymorphism $\lambda_{[]}^{\theta}$. Highlighted parts replace the old ones in $\lambda_{[]}$, rather than extensions.

B.1 Proof of the Encoding of $\lambda_{[]}^{\leqslant}$ in $\lambda_{[]}$

Lemma B.1 (Translation commutes with substitution). *If $\Delta; \Gamma, x : A \vdash M : B$ and $\Delta; \Gamma \vdash N : A$, then $[\![M[N/x]]\!] = [\![M]\!][[\![N]\!]/x]$.*

Proof. By straightforward induction on M.

x — $[\![x[N/x]]\!] = [\![N]\!] = [\![x]\!][[\![N]\!]/x]$.

$y (y \neq x)$ — $[\![y[N/x]]\!] = y = [\![y]\!][[\![N]\!]/x]$

$M_1\, M_2$ — Our goal follows from IH and definition of substitution.

$(\ell\, M')^A$ — Our goal follows from IH and definition of substitution.

$\mathbf{case}\ M'\ \{\ell_i\ x_i \mapsto N_i\}_i$ — Our goal follows from IH and definition of substitution.

$M' \rhd A$ — By IH and definition of substitution, we have $[\![(M^{[\ell_i:A_i]_i} \rhd [R])[N/x]]\!] = [\![M^{[\ell_i:A_i]_i}[N/x] \rhd [R]]\!] = \mathbf{case}\ [\![M[N/x]]\!]\ \{\ell_i\ x_i \mapsto (\ell_i\ x_i)^{[R]}\}_i = \mathbf{case}\ [\![M]\!][[\![N]\!]/x]\ \{\ell_i\ x_i \mapsto (\ell_i\ x_i)^{[R]}\}_i = (\mathbf{case}\ [\![M]\!]\ \{\ell_i\ x_i \mapsto (\ell_i\ x_i)^{[R]}\}_i)[[\![N]\!]/x] = [\![M^{[\ell_i:A_i]_i} \rhd [R]]\!][[\![N]\!]/x]$.

□

Theorem 4.1 (Type Preservation). *Every well-typed $\lambda_{[]}^{\leqslant}$ term $\Delta; \Gamma \vdash M : A$ is translated to a well-typed $\lambda_{[]}$ term $[\![\Delta]\!]; [\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]$.*

Proof. By straightforward induction on typing derivations.

T-Var — Our goal follows from $[\![x]\!] = x$ and T-Var.

T-Lam — Our goal follows from IH and T-Lam.

Syntax

$$\begin{aligned}
\mathsf{TypeScheme} \ni \tau &::= A \mid \forall \rho^K.\tau \\
\mathsf{Row} \ni R &::= \ldots \mid \rho \\
\mathsf{Term} \ni M, N &::= \ldots \mid \lambda x.M \mid \mathbf{let}\; x = M \;\mathbf{in}\; N \\
\mathsf{TyEnv} \ni \Delta &::= \ldots \mid \Delta, \rho : K \\
\mathsf{Env} \ni \Gamma &::= \boxed{\cdot \mid \Gamma, x : \tau}
\end{aligned}$$

Static Semantics

$\boxed{\Delta \vdash A : K}$

$$\textsc{K-RowVar}\quad \frac{}{\Delta, \rho : \mathsf{Row}_{\mathcal{L}} \vdash \rho : \mathsf{Row}_{\mathcal{L}}} \qquad \textsc{K-RowAll}\quad \frac{\Delta, \rho : \mathsf{Row}_{\mathcal{L}} \vdash A : \mathsf{Type}}{\Delta \vdash \forall \rho^{\mathsf{Row}_{\mathcal{L}}}.A : \mathsf{Type}}$$

$\boxed{\Delta; \Gamma \vdash M : A}$

$$\textsc{T-Lam}\quad \frac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x.M : A \to B} \qquad \textsc{T-Let}\quad \frac{\Delta; \Gamma \vdash M : \tau \qquad \Delta; \Gamma, x : \tau \vdash N : A}{\Delta; \Gamma \vdash \mathbf{let}\; x = M \;\mathbf{in}\; N : A}$$

$$\textsc{T-Inst}\quad \frac{\Delta; \Gamma \vdash M : \forall \rho^{\mathsf{Row}_{\mathcal{L}}}.\tau \qquad \Delta \vdash R : \mathsf{Row}_{\mathcal{L}}}{\Delta; \Gamma \vdash M : \tau[R/\rho]} \qquad \textsc{T-Gen}\quad \frac{\Delta, \rho : \mathsf{Row}_{\mathcal{L}}; \Gamma \vdash M : \tau \qquad \rho \notin \mathrm{ftv}(\Gamma, \Delta)}{\Delta; \Gamma \vdash M : \forall \rho^{\mathsf{Row}_{\mathcal{L}}}.\tau}$$

Dynamic Semantics

$$\beta\text{-Let} \qquad \mathbf{let}\; x = M \;\mathbf{in}\; N \leadsto_\beta N[M/x]$$

Fig. 11. Extensions and modifications to $\lambda_{\langle\rangle}$ for a calculus with rank-1 row polymorphism $\lambda^{\rho 1}_{\langle\rangle}$. Highlighted parts replace the old ones in $\lambda_{\langle\rangle}$, rather than extensions.

T-App Our goal follows from IH and T-App.

T-Inject Our goal follows from IH and T-Inject.

T-Case Our goal follows from IH and T-Case.

T-Upcast The only subtyping relation in $\lambda^{\leqslant}_{[]}$ is for variant types. Given $\Delta; \Gamma \vdash M^{[R]} \triangleright [R'] : [R']$, by $\Delta; \Gamma \vdash M : [R]$ and IH we have $[\![\Delta]\!]; [\![\Gamma]\!] \vdash [\![M]\!] : [R]$. Then, supposing $R = (\ell_i : A_i)_i$, by definition of translation, $[R] \leqslant [R']$ and T-Case we have $[\![\Delta]\!]; [\![\Gamma]\!] \vdash \mathbf{case}\; [\![M]\!]\; \{\ell_i\, x_i \mapsto (\ell_i\; x_i)^{[R']}\}_i : [R']$.

□

Theorem 4.2 (Operational Correspondence). *For the translation $[\![-]\!]$ from $\lambda^{\leqslant}_{[]}$ to $\lambda_{[]}$, we have*

Simulation *If $M \leadsto_{\beta\triangleright} N$, then $[\![M]\!] \leadsto_\beta [\![N]\!]$.*

Reflection *If $[\![M]\!] \leadsto_\beta [\![N]\!]$, then $M \leadsto_{\beta\triangleright} N$.*

Proof.

Simulation: First, we prove the base case that the whole term M is reduced, i.e. $M \leadsto_{\beta\triangleright} N$ implies $[\![M]\!] \leadsto_\beta [\![N]\!]$. The proof proceeds by case analysis on the reduction relation:

β-Lam We have $(\lambda x^A.M_1)\; M_2 \leadsto_\beta M_1[M_2/x]$. Then, (1)$[\![(\lambda x^A.M_1)\; M_2]\!] = (\lambda x^A.[\![M_1]\!])\; [\![M_2]\!] \leadsto_\beta [\![M_1]\!][[\![M_2]\!]/x] = [\![M_1[M_2/x]]\!]$, where the last equation follows from Lemma B.1.

β-Case We have $\mathbf{case}\; M'\; \{\ell_i\, x_i \mapsto N_i\}_i \leadsto_\beta N_j[M_j/x_j]$. Similar to the β-Lam case.

$\triangleright$-Upcast We have $(\ell\, M_1)^{[R]} \triangleright A \rightsquigarrow_{\triangleright} (\ell\, M_1)^A$. Supposing $R = (\ell_i : A_i)_i$, we have (2)$[\![(\ell\, M_1)^{[R]} \triangleright A]\!] = \textbf{case}\ (\ell\, [\![M_1]\!])^{[R]}\ \{\ell_i\, x_i \mapsto (\ell_i\, x_i)^A\}_i \rightsquigarrow_{\beta} (\ell\, [\![M_1]\!])^A = [\![(\ell\, M_1)^A]\!]$.

Then, we prove the full theorem by induction on M. We only need to prove the case where reduction happens in sub-terms of M.

x No reduction.

$\lambda x^A.M'$ The reduction can only happen in M'. Supposing $\lambda x^A.M' \rightsquigarrow\!\!\!\!\rightsquigarrow_{\beta\triangleright} \lambda x^A.N'$, by IH on M', we have $[\![M']\!] \rightsquigarrow\!\!\!\!\rightsquigarrow_{\beta} [\![N']\!]$, which then gives $[\![\lambda x^A.M']\!] = \lambda x^A.[\![M']\!] \rightsquigarrow\!\!\!\!\rightsquigarrow_{\beta} \lambda x^A.[\![N']\!] = [\![\lambda x^A.N']\!]$.

$M_1\, M_2$ Similar to the $\lambda x^A.M'$ case as reduction can only happen either in M_1 or M_2.

$(\ell\, M')^A$ Similar to the $\lambda x^A.M'$ case as reduction can only happen in M'.

$\textbf{case}\ M'\ \{\ell_i\, x_i \mapsto N_i\}_i$
Similar to the $\lambda x^A.M'$ case as reduction can only happen in M' or one of $(N_i)_i$.

$M' \triangleright A$ Similar to the $\lambda x^A.M'$ case as reduction can only happen in M'.

Reflection: First, we prove the base case that the whole term $[\![M]\!]$ is reduced, i.e. $[\![M]\!] \rightsquigarrow_{\beta} [\![N]\!]$ implies $M \rightsquigarrow_{\beta\triangleright} N$. The proof proceeds by case analysis on the reduction relation:

β-Lam By definition of translation, there exists M_1 and M_2 such that $M = (\lambda x^A.M_1)\ M_2$. Our goal follows from (1) and $M = (\lambda x^A.M_1)\ M_2 \rightsquigarrow_{\beta} M_1[M_2/x]$.

β-Case By definition of translation, the top-level syntax construct of M can either be **case** or upcast. Proceed by a case analysis:

- $M = \textbf{case}\ (\ell_j\, M_j)^{[R]}\ \{\ell_i\, x_i \mapsto N_i\}_i$ where $R = (\ell_i : A_i)_i$. Similar to the β-Lam case.
- $M = (\ell\, M_1)^{[R]} \triangleright A$ where $R = (\ell_i : A_i)_i$. Our goal follows from (2) and $(\ell\, M_1)^{[R]} \triangleright A \rightsquigarrow_{\triangleright} (\ell\, M_1)^A$.

Then, we prove the full theorem by induction on M. We only need to prove the case where reduction happens in sub-terms of $[\![M]\!]$.

x No reduction.

$\lambda x^A.M'$ By definition of translation, there exists N' such that $N = \lambda x^A.N'$ and $[\![M']\!] \rightsquigarrow\!\!\!\!\rightsquigarrow_{\beta} [\![N']\!]$. By IH, we have $M' \rightsquigarrow\!\!\!\!\rightsquigarrow_{\beta\triangleright} N'$, which then implies $\lambda x^A.M' \rightsquigarrow\!\!\!\!\rightsquigarrow_{\beta\triangleright} \lambda x^A.N'$.

$M_1\, M_2$ Similar to the $\lambda x^A.M'$ case as reduction can only happen either in $[\![M_1]\!]$ or $[\![M_2]\!]$.

$(\ell\, M')^A$ Similar to the $\lambda x^A.M'$ case as reduction can only happen in $[\![M']\!]$.

$\textbf{case}\ M'\ \{\ell_i\, x_i \mapsto N_i\}_i$
Similar to the $\lambda x^A.M'$ case as reduction can only happen in $[\![M']\!]$ or one of $([\![N_i]\!])_i$.

$M' \triangleright A$ Similar to the $\lambda x^A.M'$ case as reduction can only happen in $[\![M']\!]$.

□

B.2 Proof of the Encoding of $\lambda^{\leqslant}_{[\,]}$ in $\lambda^{\rho}_{[\,]}$

Lemma B.2 (Translation commutes with substitution). *If* $\Delta; \Gamma, x : A \vdash M : B$ *and* $\Delta; \Gamma \vdash N : A$*, then* $[\![M[N/x]]\!] = [\![M]\!][[\![N]\!]/x]$.

Proof. By straightforward induction on M. Only consider cases that are different from the proof of Lemma B.1.

$(\ell\, M')^{[R]}$ By IH and definition of substitution, we have $[\![(\ell\, M)^{[R]}[N/x]]\!] = [\![(\ell\, M[N/x])^{[R]}]\!] = \Lambda\rho^{\mathsf{Row}_R}.(\ell\, [\![M[N/x]]\!])^{[[\![R]\!];\rho]} = \Lambda\rho^{\mathsf{Row}_R}.(\ell\, [\![M]\!][[\![N]\!]/x])^{[[\![R]\!];\rho]} = (\Lambda\rho^{\mathsf{Row}_R}.(\ell\, [\![M]\!])^{[[\![R]\!];\rho]})[[\![N]\!]/x] = [\![(\ell\, M)^{[R]}]\!][[\![N]\!]/x]$

$\textbf{case}\ M'\ \{\ell_i\, x_i \mapsto N_i\}_i$
By an equational reasoning similar to the case of $(\ell\, M')^{[R]}$.

$M' \triangleright A$ By an equational reasoning similar to the case of $(\ell\, M')^{[R]}$.

□

Theorem 4.3 (Type Preservation). *Every well-typed $\lambda^{\leqslant}_{[\,]}$ term $\Delta; \Gamma \vdash M : A$ is translated to a well-typed $\lambda^{\rho}_{[\,]}$ term $\llbracket \Delta \rrbracket; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket A \rrbracket$.*

Proof. By induction on typing derivations.

- T-Var: Our goal follows from $\llbracket x \rrbracket = x$.
- T-Lam: Our goal follows from IH and T-Lam.
- T-App: Our goal follows from IH and T-App.
- T-Inject: By definition we have $(l : A) \in R$ implies $(l : \llbracket A \rrbracket) \in \llbracket R \rrbracket \rho$ for any ρ. Then our goal follows from IH, T-Inject and T-RowLam.
- T-Case: Our goal follows from IH and T-Case.
- T-Upcast: The only subtyping relation in $\lambda^{\leqslant}_{[\,]}$ is for variant types. Given $\Delta; \Gamma \vdash M^{[R]} \rhd [R'] : [R']$, by $\Delta; \Gamma \vdash M : [R]$ and IH we have $\llbracket \Delta \rrbracket; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket [R] \rrbracket$. Then, by definition of translation and T-RowApp we have $\llbracket \Delta \rrbracket; \llbracket \Gamma \rrbracket \vdash \llbracket M^{[R]} \rhd [R'] \rrbracket : \llbracket [R'] \rrbracket$.

□

Theorem 4.4 (Operational Correspondence). *For the translation $\llbracket - \rrbracket$ from $\lambda^{\leqslant}_{[\,]}$ to $\lambda^{\rho}_{[\,]}$, we have*

Simulation *If $M \rightsquigarrow\!\!\!\rightsquigarrow_{\beta} N$, then $\llbracket M \rrbracket \rightsquigarrow\!\!\!\rightsquigarrow^{?}_{\tau} \rightsquigarrow\!\!\!\rightsquigarrow_{\beta} \llbracket N \rrbracket$; if $M \rightsquigarrow\!\!\!\rightsquigarrow_{\rhd} N$, then $\llbracket M \rrbracket \rightsquigarrow\!\!\!\rightsquigarrow_{v} \llbracket N \rrbracket$.*

Reflection *If $\llbracket M \rrbracket \rightsquigarrow\!\!\!\rightsquigarrow^{?}_{\tau} \rightsquigarrow\!\!\!\rightsquigarrow_{\beta} \llbracket N \rrbracket$, then $M \rightsquigarrow\!\!\!\rightsquigarrow_{\beta} N$; if $\llbracket M \rrbracket \rightsquigarrow\!\!\!\rightsquigarrow_{v} \llbracket N \rrbracket$, then $M \rightsquigarrow\!\!\!\rightsquigarrow_{\rhd\blacktriangleright} N$.*

Proof.

Simulation: First, we prove the base case where the whole term M is reduced, i.e. $M \rightsquigarrow_{\beta} N$ implies $\llbracket M \rrbracket \rightsquigarrow\!\!\!\rightsquigarrow^{?}_{\tau} \rightsquigarrow\!\!\!\rightsquigarrow_{\beta} \llbracket N \rrbracket$, and $M \rightsquigarrow_{\rhd} N$ implies $\llbracket M \rrbracket \rightsquigarrow\!\!\!\rightsquigarrow_{v} \llbracket N \rrbracket$. The proof proceeds by case analysis on the reduction relation:

- β-Lam: We have $(\lambda x^{A}.M_1)\, M_2 \rightsquigarrow_{\beta} M_1[M_2/x]$. Then, (1)$\llbracket (\lambda x^{A}.M_1)\, M_2 \rrbracket = (\lambda x^{A}.\llbracket M_1 \rrbracket)\, \llbracket M_2 \rrbracket \rightsquigarrow_{\beta} \llbracket M_1 \rrbracket [\llbracket M_2 \rrbracket / x] = \llbracket M_1 [M_2/x] \rrbracket$, where the last equation follows from Lemma B.2.
- β-Case: We have $\mathbf{case}\ (\ell_j\, M_j)^{[R]}\ \{\ell_i\ x_i \mapsto N_i\} \rightsquigarrow_{\beta} N_j[M_j/x_j]$. Supposing $R = (\ell_i : A_i)_i$, we have (2)$\llbracket \mathbf{case}\ (\ell_j\, M_j)^{[R]}\ \{\ell_i\ x_i \mapsto N_i\} \rrbracket = \mathbf{case}\ (\llbracket (\ell_j\, M_j)^{[R]} \rrbracket \cdot)\ \{\ell_i\ x_i \mapsto \llbracket N_i \rrbracket\} \rightsquigarrow\!\!\!\rightsquigarrow_{\tau} \mathbf{case}\ ((\ell_j\, \llbracket M_j \rrbracket)^{\llbracket [R] \rrbracket})\ \{\ell_i\ x_i \mapsto \llbracket N_i \rrbracket\} \rightsquigarrow_{\beta} \llbracket N_j \rrbracket [\llbracket M_j \rrbracket / x_j] = \llbracket N_j [M_j/x_j] \rrbracket$, where the last equation follows from Lemma B.2.
- $\rhd$-Upcast: We have $(\ell\, M_1)^{[R]} \rhd [R'] \rightsquigarrow_{\rhd} (\ell\, M_1)^{[R']}$. We have (3)$\llbracket (\ell\, M_1)^{[R]} \rhd [R'] \rrbracket = \Lambda \rho^{\mathsf{Row}_{R'}}.\llbracket (\ell\, M_1)^{[R]} \rrbracket @ (\llbracket R' \backslash R \rrbracket; \rho) \rightsquigarrow\!\!\!\rightsquigarrow_{v} \Lambda \rho^{\mathsf{Row}_{R'}}.(\ell\, M_1)^{[\llbracket R' \rrbracket; \rho]} = \llbracket (\ell\, M_1)^{[R']} \rrbracket$.

Then, we prove the full theorem by induction on M. We only need to prove the case where reduction happens in sub-terms of M.

- x: No reduction.
- $\lambda x^{A}.M'$: The reduction can only happen in M'. Supposing $\lambda x^{A}.M' \rightsquigarrow\!\!\!\rightsquigarrow_{\beta} \lambda x^{A}.N'$, by IH on M', we have $\llbracket M' \rrbracket \rightsquigarrow\!\!\!\rightsquigarrow^{?}_{\tau} \rightsquigarrow\!\!\!\rightsquigarrow_{\beta} \llbracket N' \rrbracket$, which then gives $\llbracket \lambda x^{A}.M' \rrbracket = \lambda x^{A}.\llbracket M' \rrbracket \rightsquigarrow\!\!\!\rightsquigarrow^{?}_{\tau} \rightsquigarrow\!\!\!\rightsquigarrow_{\beta} \lambda x^{A}.\llbracket N' \rrbracket = \llbracket \lambda x^{A}.N' \rrbracket$. The same applies to the second case of the theorem.
- $(\ell\, M')^{[R]}$: Similar to the $\lambda x^{A}.M'$ case as reduction can only happen in M'.
- $M_1\ M_2$: Similar to the $\lambda x^{A}.M'$ case as reduction can only happen either in M_1 or M_2.
- $\mathbf{case}\ M'\ \{\ell_i\, x_i \mapsto N_i\}_i$: Similar to the $\lambda x^{A}.M'$ case as reduction can only happen in M' or one of $(N_i)_i$.
- $M' \rhd A$: Similar to the $\lambda x^{A}.M'$ case as reduction can only happen in M'.

Reflection: We proceed by induction on M.

- x: No reduction.
- $\lambda x^{A}.M'$: We have $\llbracket M \rrbracket = \lambda x^{\llbracket A \rrbracket}.\llbracket M' \rrbracket$. The reduction can only happen in $\llbracket M' \rrbracket$. By definition of translation, there exists N' such that $N = \lambda x^{A}.N'$ and $\llbracket M' \rrbracket \rightsquigarrow\!\!\!\rightsquigarrow^{?}_{\tau} \rightsquigarrow\!\!\!\rightsquigarrow_{\beta} \llbracket N' \rrbracket$. By IH, we

have $M' \rightsquigarrow_\beta N'$, which then implies $M \rightsquigarrow_\beta N$. The same applies to the second case of the theorem.

$M_1\ M_2$ We have $[\![M]\!] = [\![M_1]\!]\ [\![M_2]\!]$. Proceed by case analysis where the first step of reduction happens.

- Reduction happens in either $[\![M_1]\!]$ or $[\![M_2]\!]$. Similar to the $\lambda x^A.M'$ case.
- The application is reduced by β-Lam. By definition of translation, we have $M_1 = \lambda x^A.M'$. By (1), we have $[\![M]\!] \leadsto_\beta [\![M'[M_2/x]]\!]$, which then gives $N = M'[M_2/x]$. Our goal follows from $M \leadsto_\beta N$.

$(\ell\, M')^{[R]}$ We have $[\![M]\!] = \Lambda\rho^{\mathsf{Row}_R}.(\ell\,[\![M']\!])^{[[\![R]\!];\rho]}$. Similar to the $\lambda x^A.M'$ case as the reduction can only happen in $[\![M']\!]$.

case $M'\ \{\ell_i\, x_i \mapsto N_i\}_i$

We have $[\![M]\!] = \textbf{case}\ ([\![M']\!]\ \cdot)\ \{\ell_i\ x_i \mapsto [\![N_i]\!]\}_i$. Proceed by case analysis where the first step of reduction happens.

- Reduction happens in $[\![M']\!]$ or one of $[\![N_i]\!]$. Similar to the $\lambda x^A.M'$ case.
- The row type application $[\![M']\!]\cdot$ is reduced by τ-RowLam. Supposing $[\![M]\!] \rightsquigarrow_\tau N'$, by the definition of translation, because $[\![N]\!]$ must be in the codomain of the translation, we can only have $N' \leadsto_\beta [\![N]\!]$ by applying β-Case, which implies $M' = (\ell_j\, M_j)^{[R]}$. By (2), we have $[\![M]\!] \rightsquigarrow_\tau \leadsto_\beta [\![N_j[M_j/x_j]]\!]$, which then gives us $N = N_j[M_j/x_j]$. Our goal follows from $M \leadsto_\beta N$.

$M'^{[R]} \rhd [R']$

We have $[\![M]\!] = \Lambda\rho^{\mathsf{Row}_{R'}}.[\![M']\!]\ ([\![R'\backslash R]\!];\rho)$. Proceed by case analysis where the first step of reduction happens.

- Reduction happens in $[\![M']\!]$. Similar to the $\lambda x^A.M'$ case.
- The row type application $[\![M']\!]\ ([\![R'\backslash R]\!];\rho)$ is reduced by τ-RowLam. Because $[\![M']\!]$ should be a type abstraction, there are only two cases. Proceed by case analysis on M'.
  - $M' = (\ell\, M_1)^{[R]}$. By (3), we have $[\![M]\!] \rightsquigarrow_\beta [\![(\ell\, M_1)^{[R']}]\!]$, which then gives us $N = (\ell\, M_1)^{[R']}$. Our goal follows from $M \rightsquigarrow_\beta N$.
  - $M' = M_1^{[R_1]} \rhd [R]$. We have $[\![M]\!] = \Lambda\rho^{\mathsf{Row}_{R'}}.[\![M_1^{[R_1]} \rhd [R]]\!]\ ([\![R'\backslash R]\!];\rho) = \Lambda\rho^{\mathsf{Row}_{R'}}.(\Lambda\rho^{\mathsf{Row}_R}.[\![M_1]\!]@\,([\![R\backslash R_1]\!];\rho))@\,([\![R'\backslash R]\!];\rho) \rightsquigarrow_\nu \Lambda\rho^{\mathsf{Row}_{R'}}.[\![M_1]\!]@\,([\![R\backslash R_1]\!];[\![R'\backslash R]\!];\rho) = \Lambda\rho^{\mathsf{Row}_{R'}}.[\![M_1]\!]@\,([\![R'\backslash R_1]\!];\rho) = [\![M_1^{[R_1]} \rhd [R']]\!]$. By the definition of translation, we know that $N = M_1^{[R_1]} \rhd [R']$. Our goal follows from $M \rightsquigarrow_\blacktriangleright N$.

□

B.3 Proof of the Encoding $\lambda^{\leqslant}_{\langle\rangle}$ in $\lambda_{\langle\rangle}$

Lemma B.3 (Translation commutes with substitution). *If $\Delta;\Gamma, x : A \vdash M : B$ and $\Delta;\Gamma \vdash N : A$, then $[\![M[N/x]]\!] = [\![M]\!][[\![N]\!]/x]$.*

Proof. By straightforward induction on M.

x	$[\![x[N/x]]\!] = [\![N]\!] = [\![x]\!][[\![N]\!]/x]$.
$y (y \neq x)$	$[\![y[N/x]]\!] = y = [\![y]\!][[\![N]\!]/x]$
$M_1\ M_2$	Our goal follows from IH and definition of substitution.
$\langle \ell_i = M_i\rangle_i$	Our goal follows from IH and definition of substitution.
$M'.\ell$	Our goal follows from IH and definition of substitution.

$M' \rhd A$ By IH and definition of substitution, we have $[\![(M' \rhd \langle \ell_i : A_i \rangle_i)[N/x]]\!] = [\![M'[N/x] \rhd \langle \ell_i : A_i \rangle_i]\!] = \langle \ell_i = [\![M'[N/x]]\!].\ell_i \rangle_i = \langle \ell_i = [\![M']\!][[\![N]\!]/x].\ell_i \rangle_i = (\langle \ell_i = [\![M']\!].\ell_i \rangle_i)[[\![N]\!]/x] = [\![M' \rhd \langle \ell_i : A_i \rangle_i]\!][[\![N]\!]/x]$.

□

Theorem 4.5 (Type Preservation). *Every well-typed $\lambda^{\leqslant}_{\langle\rangle}$ term $\Delta; \Gamma \vdash M : A$ is translated to a well-typed $\lambda_{\langle\rangle}$ term $[\![\Delta]\!]; [\![\Gamma]\!] \vdash [\![M]\!] : [\![A]\!]$.*

Proof. By straightforward induction on typing derivations.

T-Var Our goal follows from $[\![x]\!] = x$ and T-Var.

T-Lam Our goal follows from IH and T-Lam.

T-App Our goal follows from IH and T-App.

T-Record Our goal follows from IH and T-Record.

T-Project Our goal follows from IH and T-Project.

T-Upcast The only subtyping relation in $\lambda^{\leqslant}_{\langle\rangle}$ is for record types. Given $\Delta; \Gamma \vdash M \rhd \langle R' \rangle : \langle R' \rangle$ and $\Delta; \Gamma \vdash M : \langle R \rangle$, by IH we have $[\![\Delta]\!]; [\![\Gamma]\!] \vdash [\![M]\!] : \langle R \rangle$. Then, supposing $M = \langle \ell_i = M_{\ell_i} \rangle_i$ and $R' = (\ell'_j : A_j)_j$, by definition of translation, $\langle R \rangle \leqslant \langle R' \rangle$ and T-Record we have $[\![\Delta]\!]; [\![\Gamma]\!] \vdash \langle \ell'_j = M_{\ell'_j} \rangle_j : \langle R' \rangle$.

□

Theorem 4.6 (Operational Correspondence). *For the translation $[\![-]\!]$ from $\lambda^{\leqslant}_{\langle\rangle}$ to $\lambda_{\langle\rangle}$, we have*

Simulation *If $M \rightsquigarrow_{\beta\rhd} N$, then $[\![M]\!] \rightsquigarrow^*_{\beta} [\![N]\!]$.*

Reflection *If $[\![M]\!] \rightsquigarrow_{\beta} N'$, then there exists N such that $N' \rightsquigarrow^*_{\beta} [\![N]\!]$ and $M \rightsquigarrow_{\beta\rhd} N$.*

Proof.

Simulation:

First, we prove the base case that the whole term M is reduced, i.e. $M \leadsto_{\beta\rhd} N$ implies $[\![M]\!] \leadsto^*_{\beta} [\![N]\!]$. The proof proceeds by case analysis on the reduction relation.

β-Lam We have $(\lambda x^A.M_1)\, M_2 \leadsto_{\beta} M_1[M_2/x]$. Then, (1)$[\![(\lambda x^A.M_1)\, M_2]\!] = (\lambda x^A.[\![M_1]\!])\, [\![M_2]\!] \leadsto_{\beta} [\![M_1]\!][[\![M_2]\!]/x] = [\![M_1[M_2/x]]\!]$, where the last equation follows from Lemma B.3.

β-Project We have $\langle(\ell_i = M_i)_i\rangle.\ell_j \leadsto_{\beta} M_j$. Our goal follows from (2)$[\![\langle(\ell_i = M_i)_i\rangle.\ell_j]\!] = \langle(\ell_i = [\![M_i]\!])_i\rangle.\ell_j \leadsto_{\beta} [\![M_j]\!]$.

$\rhd$-Upcast We have $\langle \ell_i = M_{\ell_i} \rangle_i \rhd \langle \ell'_j : A_j \rangle_j \leadsto_{\rhd} \langle \ell'_j = M_{\ell'_j} \rangle_j$. Our goal follows from $[\![\langle \ell_i = M_{\ell_i} \rangle_i \rhd \langle \ell'_j : A_j \rangle_j]\!] = \langle \ell'_j = [\![\langle \ell_i = M_{\ell_i} \rangle_i]\!].\ell'_j \rangle_j = \langle \ell'_j = \langle \ell_i = [\![M_{\ell_i}]\!] \rangle_i.\ell'_j \rangle_j \rightsquigarrow^*_{\beta} \langle \ell'_j = [\![M_{\ell'_j}]\!] \rangle_j$.

Then, we prove the full theorem by induction on M. We only need to prove the case where reduction happens in sub-terms of M.

x No reduction.

$\lambda x^A.M'$ The reduction can only happen in M'. Supposing $\lambda x^A.M' \rightsquigarrow_{\beta\rhd} \lambda x^A.N'$, by IH on M', we have $[\![M']\!] \rightsquigarrow^*_{\beta} [\![N']\!]$, which then gives $[\![\lambda x^A.M']\!] = \lambda x^A.[\![M']\!] \rightsquigarrow^*_{\beta} \lambda x^A.[\![N']\!] = [\![\lambda x^A.N']\!]$.

$M_1\, M_2$ Similar to the $\lambda x^A.M'$ case as reduction can only happen either in M_1 or M_2.

$\langle \ell_i = M_i \rangle_i$ Similar to the $\lambda x^A.M'$ case as reduction can only happen in one of $(M_i)_i$.

$M'.\ell$ Similar to the $\lambda x^A.M'$ case as reduction can only happen in M'.

$M' \rhd A$ Similar to the $\lambda x^A.M'$ case as reduction can only happen in M'.

Reflection: We proceed by induction on M.

x No reduction.

$\lambda x^A.M'$ We have $\llbracket M \rrbracket = \lambda x^{\llbracket A \rrbracket}.\llbracket M' \rrbracket$. The reduction can only happen in $\llbracket M' \rrbracket$. Suppose $\llbracket M \rrbracket \leadsto_\beta \lambda x^{\llbracket A \rrbracket}.N_1$. By IH on $\llbracket M' \rrbracket$, there exists N' such that $N_1 \leadsto_\beta^* \llbracket N' \rrbracket$ and $M' \leadsto_{\beta\triangleright} N'$. Our goal follows from setting N to $\lambda x^A.N'$.

$M_1\ M_2$ We have $\llbracket M \rrbracket = \llbracket M_1 \rrbracket\ \llbracket M_2 \rrbracket$. Proceed by case analysis where the reduction happens.

- Reduction happens in either $\llbracket M_1 \rrbracket$ or $\llbracket M_2 \rrbracket$. Similar to the $\lambda x^A.M'$ case.
- The application is reduced by β-Lam. By definition of translation, we have $M_1 = \lambda x^A.M'$. By (1), we have $\llbracket M \rrbracket \rightsquigarrow_\beta \llbracket M'[M_2/x] \rrbracket$. Our goal follows from setting setting N to $M'[M_2/x]$.

$\langle \ell_i = M_i \rangle_i$ We have $\llbracket M \rrbracket = \langle \ell_i = \llbracket M_i \rrbracket \rangle_i$. Similar to the $\lambda x^A.M'$ case as the reduction can only happen in one of $\llbracket M_i \rrbracket$.

$M'.\ell_j$ We have $\llbracket M \rrbracket = \llbracket M' \rrbracket.\ell_j$. Proceed by case analysis where the reduction happens.

- Reduction happens in $\llbracket M' \rrbracket$. Similar to the $\lambda x^A.M'$ case.
- The projection is reduced by β-Project. By definition of translation, we have $M' = \langle \ell_i = M_i \rangle_i$. By (2), we have $\llbracket M \rrbracket \rightsquigarrow_\beta \llbracket M_j \rrbracket$. Our goal follows from setting setting N to M_j.

$M' \triangleright \langle \ell_i : A_i \rangle_i$

We have $\llbracket M' \triangleright \langle \ell_i : A_i \rangle_i \rrbracket = \langle \ell_i = \llbracket M' \rrbracket.\ell_i \rangle_i$. Proceed by case analysis where the reduction happens.

- Reduction happens in one of $\llbracket M' \rrbracket$ in the result record. Supposing $\llbracket M \rrbracket \leadsto_\beta M_1$, and in M_1 one of $\llbracket M' \rrbracket$ is reduced to N_1. By IH on $\llbracket M' \rrbracket$, there exists N' such that $N_1 \leadsto_\beta^* \llbracket N' \rrbracket$ and $M' \leadsto_{\beta\triangleright} N'$. Thus, we can apply the reduction $\llbracket M' \rrbracket \leadsto_\beta N_1 \leadsto_\beta^* \llbracket N' \rrbracket$ to all $\llbracket M' \rrbracket$ in the result record, which gives us $\llbracket M \rrbracket \leadsto_\beta M_1 \leadsto_\beta^* \llbracket N' \triangleright \langle \ell_i : A_i \rangle_i \rrbracket$. Our goal follows from setting N to $N' \triangleright \langle \ell_i : A_i \rangle_i$ and $M' \triangleright \langle \ell_i : A_i \rangle_i \leadsto_{\beta\triangleright} N' \triangleright \langle \ell_i : A_i \rangle_i$.
- One of $\llbracket M' \rrbracket.\ell_i$ is reduced by β-Project. By the definition of translation, we know that $M' = \langle \ell'_j = M_{\ell'_j} \rangle_j$. Supposing $\llbracket M \rrbracket \leadsto_\beta M_1$, we can reduce all projection in $\llbracket M \rrbracket$, which gives us $M_1 \leadsto_\beta^* \langle \ell_i = \llbracket M_{\ell_i} \rrbracket \rangle_i = \llbracket \langle \ell_i = M_{\ell_i} \rangle_i \rrbracket$. Our goal follows from setting N to $\langle \ell_i = M_{\ell_i} \rangle_i$ and $M' \triangleright \langle \ell_i : A_i \rangle_i \rightsquigarrow_\triangleright N$.

□

B.4 Proof of the Encoding $\lambda^{\leqslant}_{\langle\rangle}$ in $\lambda^{\theta}_{\langle\rangle}$

Lemma B.4 (Translation commutes with substitution). *If $\Delta; \Gamma, x : A \vdash M : B$ and $\Delta; \Gamma \vdash N : A$, then $\llbracket M[N/x] \rrbracket = \llbracket M \rrbracket [\llbracket N \rrbracket / x]$.*

Proof. By straightforward induction on M. We only need to consider cases that are different from the proof of Lemma B.3.

$\langle \ell_i = M_i \rangle_i$ By IH and definition of substitution, we have $\llbracket \langle \ell_i = M_i \rangle_i^{\langle \ell_i : A_i \rangle_i} [N/x] \rrbracket = \llbracket \langle \ell_i = M_i[N/x] \rangle_i^{\langle \ell_i : A_i \rangle_i} \rrbracket = (\Lambda\theta_i)_i.\langle \ell_i = \llbracket M_i[N/x] \rrbracket \rangle_i^{\langle \ell_i^{\theta_i} : \llbracket A_i \rrbracket \rangle_i} = (\Lambda\theta_i)_i.\langle \ell_i = \llbracket M_i \rrbracket [\llbracket N \rrbracket / x] \rangle_i^{\langle \ell_i^{\theta_i} : \llbracket A_i \rrbracket \rangle_i} = ((\Lambda\theta_i)_i.\langle \ell_i = \llbracket M_i \rrbracket \rangle_i^{\langle \ell_i^{\theta_i} : \llbracket A_i \rrbracket \rangle_i})[\llbracket N \rrbracket / x] = \llbracket \langle \ell_i = M_i \rangle_i^{\langle \ell_i : A_i \rangle_i} \rrbracket [\llbracket N \rrbracket / x]$.

$M'.\ell$ By an equational reasoning similar to the case of $\langle \ell_i = M_i \rangle_i$.

$M' \triangleright A$ By an equational reasoning similar to the case of $\langle \ell_i = M_i \rangle_i$.

□

Theorem 4.7 (Type Preservation). *Every well-typed $\lambda^{\leqslant}_{\langle\rangle}$ term $\Delta; \Gamma \vdash M : A$ is translated to a well-typed $\lambda^{\theta}_{\langle\rangle}$ term $\llbracket \Delta \rrbracket; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket A \rrbracket$.*

Proof. By induction on typing derivations.

T-Var: Our goal follows from $\llbracket x \rrbracket = x$.

T-Lam: Our goal follows from IH and T-Lam.

T-App: Our goal follows from IH and T-App.

T-Record: Our goal follows from IH, T-Record and T-PreLam.

T-Project: Supposing $M = M'.\ell_j$ and $\Delta; \Gamma \vdash M' : \langle \ell_i : A_i \rangle_i$, by definition of translation we have $\llbracket M'.\ell_j \rrbracket = (\llbracket M' \rrbracket (P_i)_i).\ell_j$ where $P_j = \bullet$. IH on M' implies $\llbracket \Delta \rrbracket; \llbracket \Gamma \rrbracket \vdash \llbracket M' \rrbracket : (\forall \theta_i)_i.\langle \ell_i^{\theta_i} : \llbracket A_i \rrbracket \rangle_i$. Our goal follows from T-PreApp and T-Project.

T-Upcast: The only subtyping relation in $\lambda^{\leqslant}_{\langle\rangle}$ is for record types. Given $\Delta; \Gamma \vdash M^{\langle R \rangle} \triangleright [R'] : [R']$, by $\Delta; \Gamma \vdash M : \langle R \rangle$ and IH we have $\llbracket \Delta \rrbracket; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket \langle R \rangle \rrbracket$. Then, by definition of translation and T-RowApp we have $\llbracket \Delta \rrbracket; \llbracket \Gamma \rrbracket \vdash \llbracket M^{\langle R \rangle} \triangleright \langle R' \rangle \rrbracket : \llbracket \langle R' \rangle \rrbracket$.

□

Theorem 4.8 (Operational Correspondence). *The translation $\llbracket - \rrbracket$ from $\lambda^{\leqslant}_{\langle\rangle}$ to $\lambda^{\theta}_{\langle\rangle}$ has the following properties:*

Simulation *If $M \rightsquigarrow_\beta N$, then $\llbracket M \rrbracket \rightsquigarrow^*_\tau \rightsquigarrow_\beta \llbracket N \rrbracket$; if $M \rightsquigarrow_\triangleright N$, then $\llbracket M \rrbracket \rightsquigarrow^*_v \llbracket N \rrbracket$.*

Reflection *If $\llbracket M \rrbracket \rightsquigarrow^*_\tau \rightsquigarrow_\beta \llbracket N \rrbracket$, then $M \rightsquigarrow_\beta N$; if $\llbracket M \rrbracket \rightsquigarrow_v N'$, then there exists N such that $N' \rightsquigarrow^*_v \llbracket N \rrbracket$ and $M \rightsquigarrow_{\triangleright\blacktriangleright} N$.*

Proof.

Simulation: First, we prove the base case that the whole term M is reduced, i.e. $M \leadsto_\beta N$ implies $\llbracket M \rrbracket \rightsquigarrow^*_\tau \leadsto_\beta \llbracket N \rrbracket$, and $M \leadsto_\triangleright N$ implies $\llbracket M \rrbracket \rightsquigarrow^*_v \llbracket N \rrbracket$. The proof proceeds by case analysis on the reduction relation:

β-Lam: We have $(\lambda x^A.M_1)\, M_2 \leadsto_\beta M_1[M_2/x]$. Then, (1)$\llbracket (\lambda x^A.M_1)\, M_2 \rrbracket = (\lambda x^A.\llbracket M_1 \rrbracket)\, \llbracket M_2 \rrbracket \leadsto_\beta \llbracket M_1 \rrbracket [\llbracket M_2 \rrbracket / x] = \llbracket M_1[M_2/x] \rrbracket$, where the last equation follows from Lemma B.4.

β-Project: We have $\langle (\ell_i = M_i)_i \rangle.\ell_j \leadsto_\beta M_j$. By definition of translation, we have $\llbracket \langle (\ell_i = M_i)_i \rangle.\ell_j \rrbracket = (\llbracket \langle \ell_i = M_i \rangle_i \rrbracket (P_i)_i).\ell_j = (((\Lambda \theta_i)_i.\langle \ell_i^{\theta_i} = \llbracket M_i \rrbracket \rangle_i)(P_i)_i).\ell_j$, where $P_j = \bullet$ and $P_i = \circ (i \neq j)$. Applying β-PreLam, we have (2)$\llbracket \langle (\ell_i = M_i)_i \rangle.\ell_j \rrbracket \rightsquigarrow^*_\tau (\langle \ell_i^{P_i} = \llbracket M_i \rrbracket \rangle_i).\ell_j \leadsto_\beta \llbracket M_j \rrbracket$.

$\triangleright$-Upcast: We have $\langle (\ell_i = M_{\ell_i})_i \rangle^{\langle R \rangle} \triangleright \langle R' \rangle \leadsto_\triangleright \langle \ell'_j = M_{\ell'_j} \rangle_j$, where $R = (\ell_i : A_{\ell_i})_i$ and $R' = (\ell'_j : A_{\ell'_j})_j$. By definition, (3)$\llbracket \langle (\ell_i = M_{\ell_i})_i \rangle^{\langle R \rangle} \triangleright \langle R' \rangle \rrbracket = (\Lambda \theta'_j)_j.\llbracket \langle (\ell_i = M_{\ell_i})_i \rangle^{\langle R \rangle} \rrbracket\, (@\, P_i)_i = (\Lambda \theta'_j)_j.((\Lambda \theta_i)_i.\langle \ell_i = \llbracket M_{\ell_i} \rrbracket \rangle_i^{\langle \ell_i^{\theta_i} : A_{\ell_i} \rangle_i})\, (@\, P_i)_i \rightsquigarrow^*_v (\Lambda \theta'_j)_j.\langle \ell_i = \llbracket M_{\ell_i} \rrbracket \rangle_i^{\langle \ell_i^{P_i} : A_{\ell_i} \rangle_i}$, where $P_i = \circ$ when $\ell_i \notin (\ell'_j)_j$, and $P_i = \theta'_j$ when $\ell_i = \ell'_j$. By the fact that we ignore absent labels when comparing records in $\lambda^{\theta}_{\langle\rangle}$, we have (4)$(\Lambda \theta'_j)_j.\langle \ell_i = \llbracket M_{\ell_i} \rrbracket \rangle_i^{\langle \ell_i^{P_i} : A_{\ell_i} \rangle_i} = (\Lambda \theta'_j)_j.\langle \ell'_j = \llbracket M_{\ell'_j} \rrbracket \rangle^{\langle \ell_j'^{\theta'_j} : A_{\ell'_j} \rangle_j} = \llbracket \langle \ell'_j = M_{\ell'_j} \rangle_j \rrbracket$.

Then, we prove the full theorem by induction on M. We only need to prove the case where reduction happens in sub-terms of M.

x: No reduction.

$\lambda x^A.M'$: The reduction can only happen in M'. Supposing $\lambda x^A.M' \rightsquigarrow_\beta \lambda x^A.N'$, by IH on M', we have $\llbracket M' \rrbracket \rightsquigarrow^*_\tau \rightsquigarrow_\beta \llbracket N' \rrbracket$, which then gives $\llbracket \lambda x^A.M' \rrbracket = \lambda x^A.\llbracket M' \rrbracket \leadsto^*_\tau \rightsquigarrow_\beta \lambda x^A.\llbracket N' \rrbracket = \llbracket \lambda x^A.N' \rrbracket$. The same applies to the second part of the theorem.

$M_1\ M_2$: Similar to the $\lambda x^A.M'$ case as reduction can only happen either in M_1 or M_2.

$\langle \ell_i = M_i \rangle_i$: Similar to the $\lambda x^A.M'$ case as reduction can only happen in one of $(M_i)_i$.

$M'.\ell$: Similar to the $\lambda x^A.M'$ case as reduction can only happen in M'.

$M' \triangleright A$: Similar to the $\lambda x^A.M'$ case as reduction can only happen in M'.

Reflection: We proceed by induction on M.

x No reduction.

$\lambda x^A.M'$ We have $[\![M]\!] = \lambda x^{[\![A]\!]}.[\![M']\!]$. The reduction can only happen in $[\![M']\!]$. Suppose $[\![M]\!] \rightsquigarrow_\tau^* \rightsquigarrow_\beta \lambda x^{[\![A]\!]}.[\![N']\!]$. By IH on $[\![M']\!]$, $M' \rightsquigarrow_\tau^* \rightsquigarrow_\beta N'$. Our goal follows from $\lambda x^A.M' \rightsquigarrow_\tau^* \rightsquigarrow_\beta \lambda x^A.N'$. Suppose $[\![M]\!] \rightsquigarrow_\nu \lambda x^{[\![A]\!]}.N_1$. By IH on $[\![M']\!]$, there exists N' such that $N_1 \rightsquigarrow_\nu^* [\![N']\!]$ and $M' \rightsquigarrow_{\triangleright\blacktriangleright} N'$. Our goal follows from setting N to $\lambda x^A.N'$.

$M_1\ M_2$ We have $[\![M]\!] = [\![M_1]\!]\ [\![M_2]\!]$. Proceed by case analysis where the reduction happens.

- Reduction happens in either $[\![M_1]\!]$ or $[\![M_2]\!]$. Similar to the $\lambda x^A.M'$ case.
- The application is reduced by β-Lam. By definition of translation, we have $M_1 = \lambda x^A.M'$. By (1), we have $[\![M]\!] \rightsquigarrow_\beta [\![M'[M_2/x]]\!]$. Our goal follows from setting setting N to $M'[M_2/x]$.

$\langle \ell_i = M_i \rangle_i$ We have $[\![M]\!] = (\Lambda\theta_i)_i.\langle \ell_i = [\![M_i]\!] \rangle_i^{\langle \ell_i^{\theta_i} : [\![A_i]\!] \rangle_i}$. Similar to the $\lambda x^A.M'$ case as the reduction can only happen in one of $[\![M_i]\!]$.

$M'.\ell_j$ We have $[\![M]\!] = ([\![M']\!]\ (P_i)_i).\ell_j$, where $P_i = \circ$ for $i \neq j$ and $P_j = \bullet$. Proceed by case analysis where the β-reduction happens.

- Reduction happens in $[\![M']\!]$. Similar to the $\lambda x^A.M'$ case.
- The projection is reduced by β-Project*. Supposing $[\![M]\!] \rightsquigarrow_\tau^* \rightsquigarrow_\beta [\![N]\!]$, because $[\![N]\!]$ is in the codomain of the translation, the $\rightsquigarrow_\tau^*$ can only be the type applications of $(P_i)_i$ and $M' = \langle \ell_i = M_i \rangle_i$. By (2), we have $[\![M'.\ell_j]\!] \rightsquigarrow_\tau^* \rightsquigarrow_\beta [\![M_j]\!]$. Our goal follows from $M'.\ell_j \rightsquigarrow_\beta M_j$.

$M'^{\langle \ell_i : A_i \rangle_i} \triangleright \langle \ell'_j : A'_j \rangle_j$

We have $[\![M]\!] = (\Lambda\theta_j)_j.[\![M']\!]\ (@\,P_i)_i$, where $P_i = \circ$ for $\ell_i \notin (\ell'_j)_j$, and $P_i = \theta_j$ for $\ell_i = \ell'_j$. Proceed by case analysis where the reduction happens.

- Reduction happens in $[\![M']\!]$. Similar to the $\lambda x^A.M'$ case.
- The presence type application $[\![M']\!]\ @\,P_1$ is reduced by ν-PreLam. Because the top-level constructor of $[\![M']\!]$ should be type abstraction, there are two cases. Proceed by case analysis on M'.
  - $M' = \langle \ell_i = M_{\ell_i} \rangle_i$. We can reduce all presence type application of P_i. By (3) and (4), we have $[\![M]\!] \rightsquigarrow_\nu^* [\![\langle \ell'_j = M_{\ell'_j} \rangle_j]\!]$. Our goal follows from setting N to $\langle \ell'_j = M_{\ell'_j} \rangle_j$ and $M \rightsquigarrow_\triangleright N$.
  - $M' = M_1^{\langle \ell''_k : B_k \rangle_k} \triangleright \langle \ell_i : A_i \rangle_i$. We can reduce all presence type application of P_i. We have $[\![M]\!] = (\Lambda\theta_j)_j.[\![M_1 \triangleright \langle \ell_i : A_i \rangle_i]\!]\ (@\,P_i)_i =$ $(\Lambda\theta_j)_j.((\Lambda\theta_i)_i.[\![M_1]\!]\ (@\,P'_k)_k)\ (@\,P_i)_i \rightsquigarrow_\nu^* (\Lambda\theta_j)_j.[\![M_1]\!]\ (@\,Q_k)_k$, where $P'_k = \circ$ for $\ell''_k \notin (\ell_i)_i$, and $P'_k = \theta_i$ for $\ell''_k = \ell_i$. Thus, we have $Q_k = \circ$ for $\ell''_k \notin (\ell'_j)_j$, and $Q_k = \theta_j$ for $\ell''_k = \ell'_j$, which implies $[\![M_1 \triangleright \langle \ell'_j : A'_j \rangle_j]\!] = (\Lambda\theta_j)_j.[\![M_1]\!]\ (@\,Q'_k)_k$. Our goal follows from setting N to $M_1 \triangleright \langle \ell'_j : A'_j \rangle_j$ and $M \rightsquigarrow_\blacktriangleright N$.

□

C ENCODINGS, PROOFS AND DEFINITIONS IN SECTION 5

In this section, we provide the missing encodings, proofs and definitions in Section 5.

C.1 Local Term-Involved Encoding of $\lambda^{\leqslant\text{full}}_{[\,]\langle\rangle}$ in $\lambda_{[\,]\langle\rangle}$

The local term-involved encoding of $\lambda^{\leqslant\text{full}}_{[\,]\langle\rangle}$ in $\lambda_{[\,]\langle\rangle}$ [Breazu-Tannen et al. 1991; Pierce 2002] is formalised as follows.

$$
\begin{aligned}
\llbracket - \rrbracket &: \mathsf{Derivation} \longrightarrow \mathsf{Term} \\
\llbracket M^A \rhd B \rrbracket &= \llbracket A \leqslant B \rrbracket\ \llbracket M \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
\llbracket - \rrbracket &: \mathsf{Subtyping} \longrightarrow \mathsf{Term} \\
\llbracket \alpha \leqslant \alpha \rrbracket &= \lambda x^{\alpha}.x \\
\llbracket A \to B \leqslant A' \to B' \rrbracket &= \lambda f^{A \to B}.\lambda x^{A'}.\llbracket B \leqslant B' \rrbracket\ (f\ (\llbracket A' \leqslant A \rrbracket\ x)) \\
\left\llbracket \frac{\mathsf{dom}(R) \subseteq \mathsf{dom}(R') \qquad [A_i \leqslant A_i']_{(\ell_i : A_i) \in R, (\ell_i : A_i') \in R'}}{[R] \leqslant [R']} \right\rrbracket &= \lambda x^{[R]}.\mathbf{case}\ x\ \{\ell_i\ y \mapsto (\ell_i\ (\llbracket A_i \leq A_i' \rrbracket\ y))^{[R']}\} \\
\left\llbracket \frac{\mathsf{dom}(R') \subseteq \mathsf{dom}(R) \qquad [A_i \leqslant A_i']_{(\ell_i : A_i) \in R, (\ell_i : A_i') \in R'}}{\langle R \rangle \leqslant \langle R' \rangle} \right\rrbracket &= \lambda x^{\langle R \rangle}.\langle \ell_i = \llbracket A_i \leq A_i' \rrbracket\ x.\ell_i \rangle
\end{aligned}
$$

C.2 Dynamic Semantics of $\lambda^{\leqslant\text{full}}_{[\,]\langle\rangle}$

In addition to the erasure semantics, the other style of dynamic semantics of $\lambda^{\leqslant\text{full}}_{[\,]\langle\rangle}$ is given by extending the operational semantics rules with the following four upcast rules.

$$
\begin{array}{ll}
\rhd\text{-Var} & M \rhd \alpha \rightsquigarrow_{\rhd} M \\
\rhd\text{-Lam} & (\lambda x^A.M) \rhd A' \to B' \rightsquigarrow_{\rhd} \lambda y^{A'}.(M[(y \rhd A)/x] \rhd B') \\
\rhd\text{-Variant} & (\ell_j\ M)^A \rhd [\ell_i : A_i]_i \rightsquigarrow_{\rhd} (\ell_j\ (M \rhd A_j))^{[\ell_i : A_i]_i} \\
\rhd\text{-Record} & \langle \ell_i = M_{\ell_i} \rangle_i \rhd \langle \ell_j' : A_j \rangle_j \rightsquigarrow_{\rhd} \langle \ell_j' = M_{\ell_j'} \rhd A_j \rangle_j
\end{array}
$$

We show that there is a correspondence between these two styles of dynamic semantics of $\lambda^{\leqslant\text{full}}_{[\,]\langle\rangle}$. We first give a preorder $M \sqsubseteq N$ on terms of the untyped $\lambda_{[\,]\langle\rangle}$ which allows records in M to contain more elements than those in N, because the erasure semantics does not truly perform upcasts. The full definition is shown in Figure 12.

$$
\frac{\{\ell_j'\}_j \subseteq \{\ell_i\}_i \qquad [M_i \sqsubseteq N_j]_{\ell_i = \ell_j'}}{\langle \ell_i = M_i \rangle_i \sqsubseteq \langle \ell_j' = N_j \rangle_j}
\qquad
x \sqsubseteq x
\qquad
\frac{M \sqsubseteq M'}{\lambda x.M \sqsubseteq \lambda x.M'}
\qquad
\frac{M \sqsubseteq M' \qquad N \sqsubseteq N'}{M\ N \sqsubseteq M'\ N'}
$$

$$
\frac{M \sqsubseteq M'}{\ell\ M \sqsubseteq \ell\ M'}
\qquad
\frac{M \sqsubseteq M' \qquad [N_i \sqsubseteq N_i']_i}{\mathbf{case}\ M\ \{\ell_i\ x_i \mapsto N_i\}_i \sqsubseteq \mathbf{case}\ M'\ \{\ell_i\ x_i \mapsto N_i'\}_i}
\qquad
\frac{M \sqsubseteq M'}{M.\ell \sqsubseteq M'.\ell}
$$

Fig. 12. The preorder $\sqsubseteq$ of untyped $\lambda_{[\,]\langle\rangle}$.

The correspondence is given by the following theorem.

Theorem C.1 (Operational Correspondence). *Given a well-typed term M in $\lambda^{\leqslant\text{full}}_{[\,]\langle\rangle}$ and a term M' in untyped $\lambda_{[\,]\langle\rangle}$ with $M' \sqsubseteq \mathsf{erase}(M)$, we have:*

Simulation *If $M \rightsquigarrow_{\beta} N$, then there exists N' such that $N' \sqsubseteq \mathsf{erase}(N)$ and $M' \rightsquigarrow_{\beta} N'$; if $M \rightsquigarrow_{\rhd} N$, then $M' \sqsubseteq \mathsf{erase}(N)$.*

Reflection *If $M' \rightsquigarrow_{\beta} N'$, then there exists N such that $N' \sqsubseteq \mathsf{erase}(N)$ and $M \rightsquigarrow_{\rhd}^{*} \rightsquigarrow_{\beta} N$.*

To prove it, we need two lemmas.

Lemma C.2 (Erasure commutes with substitution). *If $\Delta; \Gamma, x : A \vdash M : B$ and $\Delta; \Gamma \vdash N : A$, then for $M' \sqsubseteq \mathsf{erase}(M)$ and $N' \sqsubseteq \mathsf{erase}(N)$, we have $M'[N'/x] \sqsubseteq \mathsf{erase}(M[N/x])$.*

Proof. By straightforward induction on M. □

Lemma C.3 (Upcasts shrink terms). *For any $M \rhd A \rightsquigarrow_{\rhd} N$ in $\lambda^{\leqslant\text{full}}_{[\,]\langle\rangle}$, we have $\mathsf{erase}(M) \sqsubseteq \mathsf{erase}(N)$.*

Proof. By definition of erase$(-)$ and $\rightsquigarrow_{\triangleright}$. □

Then, we give the proof of Theorem C.1.

Proof.

Simulation: We proceed by induction on M.

x No reduction.

$\lambda x^A.M_1$ Supposing $M' = \lambda x.M_1'$, by $M' \sqsubseteq \text{erase}(M)$ we have $M_1' \sqsubseteq \text{erase}(M_1)$. The reduction must happen in M_1. Our goal follows from the IH on M_1.

$M_1\ M_2$ Supposing $M' = M_1'\ M_2'$, by $M' \sqsubseteq \text{erase}(M)$ we have $M_1' \sqsubseteq \text{erase}(M_1)$ and $M_2' \sqsubseteq \text{erase}(M_2)$. We proceed by case analysis where the reduction happens.

- The reduction happens in either M_1 or M_2. Our goal follows from the IH.
- The reduction reduces the top-level function application. Supposing $M_1 = \lambda x^A.M_3$ and $M_1' = \lambda x.M_3'$ with $M_3' \sqsubseteq \text{erase}(M_3)$, we have $(\lambda x^A.M_3)\ M_2 \rightsquigarrow_\beta M_3[M_2/x]$ and $(\lambda x^A.M_3')\ M_2' \rightsquigarrow_\beta M_3'[M_2'/x]$. Our goal follows from Lemma C.2.

$N.\ell_k$ Supposing $M' = N'.\ell_k$, by $M' \sqsubseteq \text{erase}(M)$ we have $N' \sqsubseteq \text{erase}(N')$. We proceed by case analysis where the reduction happens.

- The reduction happens in N. Our goal follows from the IH on N.
- The reduction reduces the top-level projection. Supposing $N = \langle \ell_i = M_i \rangle_i$ and $N' = \langle \ell_j' = M_j' \rangle_j$ with $\{\ell_j'\}_j \subseteq \{\ell_i\}_i$ and $(M_j' \sqsubseteq \text{erase}(M_i))_{\ell_i = \ell_j'}$, we have $N.\ell_k \rightsquigarrow_\beta M_k$ and $N'.\ell_k \rightsquigarrow_\beta M_n'$ where $\ell_k = \ell_n'$. Our goal follows from $M_n' \sqsubseteq \text{erase}(M_k)$.

$\langle \ell_i = M_i \rangle_i$ The reduction must happen in one of the M_i. Our goal follows from the IH.

$M_1 \triangleright A$ For the β-reduction, it must happen in M_1. Our goal follows from the IH. For the upcast reduction, by $M' \sqsubseteq \text{erase}(M)$ we have $M' \sqsubseteq \text{erase}(M_1)$. By Lemma C.3, we have $M' \sqsubseteq \text{erase}(M_1) \sqsubseteq \text{erase}(N)$.

Reflection: We proceed by induction on M'.

x No reduction.

$\lambda x.M_1'$ By $M' \sqsubseteq \text{erase}(M)$, we know that there exists $\lambda x^A.M_1$ such that $M \rightsquigarrow_{\triangleright}^* \lambda x^A.M_1$. By Lemma C.3, $\text{erase}(M) \sqsubseteq \text{erase}(\lambda x^A.M_1)$. Then, by $M' \sqsubseteq \text{erase}(M)$ and transitivity, we have $M_1' \sqsubseteq \text{erase}(M_1)$. The β-reduction must happen in M_1'. Our goal follows from the IH on M_1'.

$M_1'\ M_2'$ By $M' \sqsubseteq \text{erase}(M)$, we know that there exists $M_1\ M_2$ such that $M \rightsquigarrow_{\triangleright}^* M_1\ M_2$. By Lemma C.3 and $M' \sqsubseteq \text{erase}(M)$, we have $M_1' \sqsubseteq \text{erase}(M_1)$ and $M_2' \sqsubseteq \text{erase}(M_2)$. We proceed by case analysis where the reduction happens.

- The reduction happens in either M_1' or M_2'. Our goal follows from the IH.
- The reduction reduces the top-level function application. Supposing $M_1' = \lambda x.M_3'$, by $M_1' \sqsubseteq \text{erase}(M_1)$, we know that there exists $\lambda x^A.M_3$ such that $M_1 \rightsquigarrow_{\triangleright}^* \lambda x^A.M_3$. Thus, $M_1\ M_2 \rightsquigarrow_{\triangleright}^* \rightsquigarrow_\beta M_3[M_2/x]$ and $M_1'\ M_2' \rightsquigarrow_\beta M_3'[M_2'/x]$. By Lemma C.3, we have $M_1' \sqsubseteq \text{erase}(M_1) \sqsubseteq \text{erase}(\lambda x^A.M_3)$, which implies $M_3' \sqsubseteq \text{erase}(M_3)$. Our goal follows from Lemma C.2.

$N'.\ell_k$ By $M' \sqsubseteq \text{erase}(M)$, we know that there exists $N.\ell_k$ such that $M \rightsquigarrow_{\triangleright}^* N.\ell_k$. By Lemma C.3 and $M' \sqsubseteq \text{erase}(M)$, we have $N' \sqsubseteq \text{erase}(N)$. We proceed by case analysis where the reduction happens.

- The reduction happens in N'. Our goal follows from the IH on N.
- The reduction reduces the top-level projection. Supposing $N' = \langle \ell_j' = M_j' \rangle_j$, by $N' \sqsubseteq \text{erase}(N)$, we know that there exists $\langle \ell_i = M_i \rangle_i$ such that $N \rightsquigarrow_{\triangleright}^* \langle \ell_i = M_i \rangle_i$. Thus, $N.\ell_k \rightsquigarrow_{\triangleright}^* \rightsquigarrow_\beta M_k$ and $N'.\ell_k \rightsquigarrow_\beta M_n'$ where $\ell_n' = \ell_k$. By Lemma C.3, we have

$\mathsf{erase}(N) \sqsubseteq \mathsf{erase}(\langle \ell_i = M_i \rangle_i)$. We can further conclude that $M'_n \sqsubseteq \mathsf{erase}(M_k)$ from $N' \sqsubseteq \mathsf{erase}(N)$. □

C.3 Proof of the Encoding of $\lambda_{\langle\rangle}^{\leqslant\mathrm{co}}$ in $\lambda_{\langle\rangle}^{\theta}$

Lemma C.4 (Upcast Translation). *If $A \leqslant B$, then $\forall \overline{\theta}.\llbracket A, \overline{P} \rrbracket = \llbracket B \rrbracket$ for $(\overline{\theta}, \overline{P}) = \llparenthesis \theta, A \leqslant B \rrparenthesis$.*

Proof. By a straightforward induction on the definition of $\llparenthesis \theta, A \leqslant B \rrparenthesis$. □

Theorem 5.1 (Type Preservation). *Every well-typed $\lambda_{\langle\rangle}^{\leqslant\mathrm{co}}$ term $\Delta; \Gamma \vdash M : A$ is translated to a well-typed $\lambda_{\langle\rangle}^{\theta}$ term $\llbracket \Delta \rrbracket; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket A \rrbracket$.*

Proof. By induction on typing derivations.

T-Var Our goal follows from $\llbracket x \rrbracket = x$.

T-Lam By the IH on $\Delta; \Gamma, x : A \vdash M : B$, we have

$$\Delta; \llbracket \Gamma \rrbracket, x : \llbracket A \rrbracket \vdash \llbracket M \rrbracket : \llbracket B \rrbracket$$

Let $\overline{\theta} = \llparenthesis \theta, B \rrparenthesis$. By T-PreApp and context weakening, we have

$$\Delta, \overline{\theta}; \llbracket \Gamma \rrbracket, x : \llbracket A \rrbracket \vdash \llbracket M \rrbracket\ \overline{\theta} : \llbracket B, \overline{\theta} \rrbracket$$

Notice that we always assume variable names in the same context are unique, so we do not need to worry that $\overline{\theta}$ conflicts with Δ. Then, by T-Lam, we have

$$\Delta, \overline{\theta}; \llbracket \Gamma \rrbracket \vdash \lambda x^{\llbracket A \rrbracket}.\llbracket M \rrbracket\ \overline{\theta} : \llbracket A \rrbracket \to \llbracket B, \overline{\theta} \rrbracket$$

Finally, by T-PreLam, we have

$$\Delta; \llbracket \Gamma \rrbracket \vdash \Lambda \overline{\theta}.\lambda x^{\llbracket A \rrbracket}.\llbracket M \rrbracket\ \overline{\theta} : \forall \overline{\theta}.\llbracket A \rrbracket \to \llbracket B, \overline{\theta} \rrbracket$$

Our goal follows from $\llbracket A \to B \rrbracket = \forall \overline{\theta}.\llbracket A \rrbracket \to \llbracket B, \overline{\theta} \rrbracket$.

T-App Similar to the T-Lam case. Our goal follows from IH, T-App, T-PreApp and T-PreLam.

T-Record Similar to the T-Lam case. Our goal follows from IH, T-Record, T-PreApp and T-PreLam.

T-Project Given the derivation of $\Delta; \Gamma \vdash M.\ell_j A_j$, by the IH on $\Delta; \Gamma \vdash M : \langle \ell_i : A_i \rangle_i$, we have

$$\Delta; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket \langle \ell_i : A_i \rangle_i \rrbracket$$

Let $P_i = \circ (i \neq j), P_j = \bullet, \overline{\theta} = \llparenthesis \theta, A_j \rrparenthesis, \overline{P}_i = \llparenthesis \circ, A_i \rrparenthesis$. By T-PreApp and context weakening, we have

$$\Delta, \overline{\theta}; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket\ (P_i)_i\ (\overline{P}_i)_{i<j}\ \overline{\theta}\ (\overline{P}_i)_{j<i} : \langle R \rangle$$

where $\ell_j : \llbracket A_j, \overline{\theta} \rrbracket \in R$ by the definition of translations and the canonical order. Then, by T-Proj, we have

$$\Delta, \overline{\theta}; \llbracket \Gamma \rrbracket \vdash (\llbracket M \rrbracket\ (P_i)_i\ (\overline{P}_i)_{i<j}\ \overline{\theta}\ (\overline{P}_i)_{j<i}).\ell_j : \llbracket A_j, \overline{\theta} \rrbracket$$

Finally, by T-PreLam, we have

$$\Delta; \llbracket \Gamma \rrbracket \vdash (\llbracket M \rrbracket\ (P_i)_i\ (\overline{P}_i)_{i<j}\ \overline{\theta}\ (\overline{P}_i)_{j<i}).\ell_j : \forall \overline{\theta}.\llbracket A_j, \overline{\theta} \rrbracket$$

Our goal follows from $\llbracket A_j \rrbracket = \forall \overline{\theta}.\llbracket A_j, \overline{\theta} \rrbracket$ where $\overline{\theta} = \llparenthesis \theta, A_j \rrparenthesis$.

T-Upcast Given the derivation of $\Delta; \Gamma \vdash M \rhd B : B$, by the IH on $\Delta; \Gamma \vdash M : A$, we have

$$\Delta; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket A \rrbracket$$

Let $(\overline{\theta}, \overline{P}) = \llparenthesis \theta, A \leqslant B \rrparenthesis$. By T-PreApp and context weakening, we have

$$\Delta, \overline{\theta}; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket\ \overline{P} : \llbracket A, \overline{P} \rrbracket$$

Then, by T-PreLam, we have

$$\Delta; [\![\Gamma]\!] \vdash \Lambda\overline{\theta}.[\![M]\!]\ \overline{P} : \forall\overline{\theta}.[\![A, \overline{P}]\!]$$

By Lemma C.4, we have $[\![B]\!] = \forall\overline{\theta}.[\![A, \overline{P}]\!]$.

□

D THE PROOF IN SECTION 6

In this section, we provide the missing proof in Section 6.

D.1 Proof of encoding $\lambda^{\leqslant\text{full}}_{\langle\rangle 2}$ using $\lambda^{\rho 1}_{\langle\rangle}$

Theorem 6.2 (Weak Type Preservation). *Every well-typed $\lambda^{\leqslant\text{full}}_{\langle\rangle 2}$ term $\Delta; \Gamma \vdash M : A$ is translated to a well-typed $\lambda^{\rho 1}_{\langle\rangle}$ term $[\![\Delta; \Gamma]\!] \vdash [\![M]\!] : \tau$ for some $A' \leqslant A$ and $\tau \leqslant [\![A']\!]$.*

Proof. As shown in Section 6, we only need to prove that $\Delta; \Gamma \vdash M : A$ in $\lambda^{\leqslant\text{afull}}_{\langle\rangle}$ implies $[\![\Delta; \Gamma]\!] \vdash [\![M]\!] : \tau$ for some $\tau \leqslant [\![A]\!]$ in $\lambda^{\rho 1}_{\langle\rangle}$. We proceed by induction on the typing derivations in $\lambda^{\leqslant\text{afull}}_{\langle\rangle}$.

T-Var Our goal follows directly from the definition of translations.

T-Lam Given the derivation of $\Delta; \Gamma \vdash \lambda a^A.M : A \to B$, by the IH on $\Delta; \Gamma, a : A \vdash M : B$, we have

$$\Delta, \mathsf{ftv}([\![\Gamma]\!]), \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*; \Gamma, a : [\![A, \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*]\!]^* \vdash [\![M]\!] : \tau_B$$

for some $\tau_B \leqslant [\![B]\!]$. Supposing $\tau_B = \forall\overline{\rho}_B.B'$, by T-Inst and environment weakening, we have [3]

$$\Delta, \mathsf{ftv}([\![\Gamma]\!]), \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*, \overline{\rho}_B; \Gamma, a : [\![A, \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*]\!]^* \vdash [\![M]\!] : B'$$

Then, by T-Lam, we have

$$\Delta, \mathsf{ftv}([\![\Gamma]\!]), \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*, \overline{\rho}_B; \Gamma \vdash \lambda a.[\![M]\!] : [\![A, \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*]\!]^* \to B'$$

Finally, by T-Gen, we have

$$\Delta, \mathsf{ftv}([\![\Gamma]\!]); \Gamma \vdash \lambda a.[\![M]\!] : \forall \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*\ \overline{\rho}_B.[\![A, \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*]\!]^* \to B'$$

By definition, we have $[\![A \to B]\!] = \forall\overline{\rho}_1\overline{\rho}_2.[\![A, \overline{\rho}_1]\!]^* \to [\![B, \overline{\rho}_2]\!]$, where $\overline{\rho}_1 = \llparenthesis \rho_1, A \rrparenthesis^*$, $\overline{\rho}_2 = \llparenthesis \rho_2, B \rrparenthesis$. It is easy to check that $\forall \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*\ \overline{\rho}_B.[\![A, \llparenthesis \rho_{|\Gamma|}, A \rrparenthesis^*]\!]^* \to B' \leqslant [\![A \to B]\!]$ under α-renaming.

T-AppSub Given the derivation of $\Delta; \Gamma \vdash M\,N : B$, by the IH on $\Delta; \Gamma \vdash M : A \to B$, we have

$$[\![\Delta; \Gamma]\!] \vdash [\![M]\!] : \tau_1$$

for some $\tau_1 \leqslant [\![A \to B]\!]$. By the IH on $\Delta; \Gamma \vdash B : A_2$, we have

$$[\![\Delta; \Gamma]\!] \vdash [\![N]\!] : \tau_2$$

for some $\tau_2 \leqslant [\![A_2]\!]$. We have $\mho^2(A \to B)$, which implies $\mho^1(A)$. Then, $A_2 \leqslant A$ gives us $\mho^1(A_2)$, which further implies that $[\![A_2]\!] = A_2$ and τ_2 is not polymorphic. Thus, we

[3]We always assume type variables in type environments have different names, and we omit kinds when they are easy to reconstruct from the context.

have $\tau_2 \preccurlyeq \llbracket A_2 \rrbracket = A_2 \preccurlyeq A$. Notice that given $A \preccurlyeq \_ \preccurlyeq B$ with $\mho^1(B)$, we can always construct $\overline{R}$ with $\llbracket B, \overline{R} \rrbracket^* = A$, by $\llparenthesis A \preccurlyeq\preccurlyeq B \rrparenthesis$ defined as follows.

$$\begin{aligned}
\llparenthesis - \rrparenthesis \ &: \ (\mathsf{Type} \preccurlyeq\preccurlyeq \mathsf{Type}) \to (\overline{\mathsf{Row}}) \\
\llparenthesis \alpha \preccurlyeq\preccurlyeq \alpha \rrparenthesis &= (\cdot, \cdot) \\
\llparenthesis A \to B \preccurlyeq\preccurlyeq A \to B' \rrparenthesis &= \llparenthesis B \preccurlyeq\preccurlyeq B' \rrparenthesis \\
\llparenthesis \langle (\ell_i : A_i)_i \rangle \preccurlyeq\preccurlyeq \langle (\ell'_j : A'_j) \rangle \rrparenthesis &= (\ell_k : A_k)_{k \in \{\ell_i\}_i \backslash \{\ell'_j\}_j} \ \llparenthesis A_i \preccurlyeq\preccurlyeq A'_j \rrparenthesis_{\ell_i = \ell'_j} \\
\llparenthesis \langle (\ell_i : A_i)_i ; \rho \rangle \preccurlyeq\preccurlyeq \langle (\ell'_j : A'_j) \rangle \rrparenthesis &= ((\ell_k : A_k)_{k \in \{\ell_i\}_i \backslash \{\ell'_j\}_j} ; \rho) \ \llparenthesis A_i \preccurlyeq\preccurlyeq A'_j \rrparenthesis_{\ell_i = \ell'_j}
\end{aligned}$$

Let $\overline{R} = \llparenthesis \tau_2 \preccurlyeq\preccurlyeq A \rrparenthesis$. We have $\llbracket A, \overline{R} \rrbracket^* = \tau_2$. Suppose $\tau_1 = \forall \overline{\rho}. A' \to B'$. By definition, we have $\llbracket A \to B \rrbracket = \forall \overline{\rho}_1 \overline{\rho}_2 . \llbracket A, \overline{\rho}_1 \rrbracket^* \to \llbracket B, \overline{\rho}_2 \rrbracket$, where $\overline{\rho}_1 = \llparenthesis \rho_1, A \rrparenthesis^*$, $\overline{\rho}_2 = \llparenthesis \rho_2, B \rrparenthesis$. By $\tau_1 \preccurlyeq \llbracket A \to B \rrbracket$, we have $A' = \llbracket A, \overline{\rho}_1 \rrbracket^*$, $B' \preccurlyeq \llbracket B, \overline{\rho}_2 \rrbracket$ and $\overline{\rho} = \overline{\rho}_1 \overline{\rho}_2$ after α-renaming. By T-Inst and environment weakening, we have

$$\Delta, \mathsf{ftv}(\llbracket \Gamma \rrbracket), \overline{\rho}_2 ; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \llbracket A, \overline{R} \rrbracket^* \to B'$$

Notice that $\llbracket A, \overline{R} \rrbracket^* = \tau_2$. We can then apply T-App and environment weakening, which gives us

$$\Delta, \mathsf{ftv}(\llbracket \Gamma \rrbracket), \overline{\rho}_2 ; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket \, \llbracket N \rrbracket : B'$$

Finally, by T-Gen, we have

$$\Delta, \mathsf{ftv}(\llbracket \Gamma \rrbracket) ; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket \, \llbracket N \rrbracket : \forall \overline{\rho}_2 . B'$$

The condition $\forall \overline{\rho}_2 . B' \preccurlyeq \llbracket B \rrbracket$ holds obviously.

T-Record Our goal follows from the IH and a sequence of applications of T-Inst, T-Record, and T-Gen similar to the previous cases.

T-Project Our goal follows from the IH and a sequence of applications of T-Inst, T-Project, and T-Gen similar to the previous cases.

T-Let Given the derivation of $\Delta ; \Gamma \vdash \mathbf{let}\, x = M \,\mathbf{in}\, N$, by the IH on $\Delta ; \Gamma \vdash M : A$, we have

$$\Delta, \mathsf{ftv}(\llbracket \Gamma \rrbracket) ; \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket : \tau_1$$

for some $\tau_1 \preccurlyeq \llbracket A \rrbracket$. By the IH on $\Delta ; \Gamma, x : A \vdash N : B$, we have

$$\Delta, \mathsf{ftv}(\llbracket \Gamma \rrbracket) ; \llbracket \Gamma \rrbracket, x : \llbracket A \rrbracket \vdash \llbracket N \rrbracket : \tau_2$$

for some $\tau_2 \preccurlyeq \llbracket B \rrbracket$. By another straightforward induction on the typing derivations, we can show that $\Delta ; \Gamma, x : \tau_1 \vdash M : \tau_2$ implies $\Delta ; \Gamma, x : \tau'_1 \vdash M : \tau'_2$ for $\tau'_1 \preccurlyeq \tau_1$ and $\tau'_2 \preccurlyeq \tau_2$. Thus, we have

$$\Delta, \mathsf{ftv}(\llbracket \Gamma \rrbracket) ; \llbracket \Gamma \rrbracket, x : \tau_1 \vdash \llbracket N \rrbracket : \tau'_2$$

for some $\tau'_2 \preccurlyeq \tau_2 \preccurlyeq \llbracket B \rrbracket$. Then, by T-Let, we have

$$\Delta, \mathsf{ftv}(\llbracket \Gamma \rrbracket) ; \llbracket \Gamma \rrbracket \vdash \mathbf{let}\, x = \llbracket M \rrbracket \,\mathbf{in}\, \llbracket N \rrbracket : \tau'_2$$

with $\tau'_2 \preccurlyeq \llbracket B \rrbracket$.

□

E PROOFS OF NON-EXISTENCE RESULTS

In this section, we give the proofs of non-existence results in Section 4 and Section 5.

E.1 Non-Existence of Type-Only Encodings of $\lambda^{\leqslant}_{\langle\rangle}$ in $\lambda^{\rho}_{\langle\rangle}$ and $\lambda^{\leqslant}_{[]}$ in $\lambda^{\theta}_{[]}$

Theorem 4.9. *There exists no global type-only encoding of $\lambda^{\leqslant}_{\langle\rangle}$ in $\lambda^{\rho}_{\langle\rangle}$, and no global type-only encoding of $\lambda^{\leqslant}_{[]}$ in $\lambda^{\theta}_{[]}$.*

Proof. We provide three proofs of this theorem, the first one is based on the type preservation property, the second one is based on the compositionality of translations, and the third one carefully avoids using the type preservation and compositionality. The point of multiple proofs is to show that the non-existence of the encoding of $\lambda^{\leqslant}_{\langle\rangle}$ in $\lambda^{\rho}_{\langle\rangle}$ is still true even if we relax the condition of type preservation and compositionality, which emphasises the necessity of the restrictions in Section 6.
Proof 1:

We assume that $\Delta = \alpha_0$ and $\Gamma = y : \alpha_0$ when environments are omitted.

Consider $\langle\rangle$ and $\langle \ell = y\rangle \rhd \langle\rangle$. By the fact that $[\![-]\!]$ is type-only, we have $[\![\langle\rangle]\!] = \Lambda\overline{\alpha}.\langle\rangle$ and $[\![\langle \ell = y\rangle \rhd \langle\rangle]\!] = \Lambda\overline{\beta}.[\![\langle \ell = y\rangle]\!]\ \overline{B} = \Lambda\overline{\beta}.(\Lambda\overline{\gamma}.\langle \ell = \Lambda\overline{\gamma}'.y\rangle)\ \overline{B}$. Thus, $[\![\langle \ell = y\rangle \rhd \langle\rangle]\!]$ has type $\forall\overline{\alpha}'.\langle \ell : \forall\overline{\gamma}'.\alpha_0\rangle$ for some $\overline{\alpha}'$.

By type preservation, the translated results should have the same type, which implies $\forall\overline{\alpha}.\langle\rangle = \forall\overline{\alpha}'.\langle \ell : \forall\overline{\gamma}'.\alpha_0\rangle$. Thus, we have the equation $\langle\rangle = \langle \ell : \forall\overline{\gamma}'.\alpha_0\rangle$, which leads to a contradiction as the right-hand side has an extra label ℓ and we do not have presence types to remove labels.

Similarly, we can prove the theorem for variants by considering $(\ell_1\ y)^{[\ell_1:\alpha_0;\ell_2:\alpha_0]}$ and $(\ell_1\ y)^{[\ell_1:\alpha_0]} \rhd [\ell_1 : \alpha_0; \ell_2 : \alpha_0]$. The key point is that ℓ_2 is arbitrarily chosen, so for the translation of $(\ell_1\ y)^{[\ell_1:\alpha_0]}$ we cannot guarantee that ℓ_2 appears in its type, and presence polymorphism does not give us the ability to add new labels to row types.
Proof 2:

We assume that $\Delta = \alpha_0$ and $\Gamma = y : \alpha_0$ when environments are omitted.

Consider the function application $M\ N$ where $M = \lambda x^{\langle\rangle}.\langle\rangle$ and $N = \langle \ell = y\rangle \rhd \langle\rangle$. By the type-only property, we have

$$[\![\lambda x^{\langle\rangle}.\langle\rangle]\!] = \Lambda\overline{\alpha}_1.\lambda x^{A_1}.\Lambda\overline{\beta}_1.\langle\rangle\ B_1$$

for some $\overline{\alpha}_1, \overline{\beta}_1, A_1$ and B_1. By Proof 1, we have

$$[\![\langle \ell = y\rangle \rhd \langle\rangle]\!] = \Lambda\overline{\alpha}_2.\langle \ell = \Lambda\overline{\beta}_2.y\rangle$$

for some $\overline{\alpha}_2$ and $\overline{\beta}_2$. Then, by the type-only property, we have

$$[\![(\lambda x^{\langle\rangle}.\langle\rangle)\ (\langle \ell = y\rangle \rhd \langle\rangle)]\!] = \Lambda\overline{\alpha}.([\![\lambda x^{\langle\rangle}.\langle\rangle]\!]\ \overline{A})\ (\Lambda\overline{\beta}.[\![\langle \ell = y\rangle \rhd \langle\rangle]\!]\ \overline{B})\ \overline{C}$$

for some $\overline{\alpha}, \overline{\beta}, \overline{A}, \overline{B}$ and $\overline{C}$. As we only have row polymorphism, the type application of $\overline{B}$ cannot remove the label ℓ from the type of $[\![N]\!]$. Since ℓ is arbitrarily chosen, it can neither be already in the type of $[\![M]\!]$. By definition, a compositional translation can only use the type information of M and N, which contains nothing about the label ℓ. Thus, the label ℓ can neither be in $\overline{A}$, which further implies that the $[\![M\ N]\!]$ is not well-typed as the T-App must fail. Contradiction.
Proof 3:

Consider three functions $f_1 = \lambda x^{\langle\rangle}.x$, $f_2 = \lambda x^{\langle\rangle}.\langle\rangle$, and $g = \lambda f^{\langle\rangle\to\langle\rangle}.\langle\rangle$. By the type-only property, we have

$$\begin{aligned}
[\![f_1]\!] &= \Lambda\overline{\alpha}_1.\lambda x^{A_1}.\Lambda\overline{\beta}_1.x\ \overline{B}_1 &&: \forall\overline{\alpha}_1.A_1 \to \forall\overline{\beta}_1.A_1' \\
[\![f_2]\!] &= \Lambda\overline{\alpha}_2.\lambda x^{A_2}.\Lambda\overline{\beta}_2.\langle\rangle &&: \forall\overline{\alpha}_2.A_2 \to \forall\overline{\beta}_2.\langle\rangle \\
[\![g]\!] &= \Lambda\overline{\alpha}_3.\lambda f^{A_3}.\Lambda\overline{\beta}_3.\langle\rangle &&: \forall\overline{\alpha}_3.A_3 \to \forall\overline{\beta}_3.\langle\rangle
\end{aligned}$$

where $A_1' = A_1''[\overline{B}_1/\overline{\alpha}_1']$ and $A_1 = \forall\overline{\alpha}_1'.A_1''$.

If there is some variable $\alpha_1' \in \overline{\alpha}_1$ appears in A_1, then it must also appear in A_1' as we have no way to remove it by the substitution $[\overline{B}_1/\overline{\alpha}']$. Thus, A_3 should be of shape $\forall\overline{\alpha}.A \to \forall\overline{\beta}.A'$ where A'

contains some variable $\alpha' \in \overline{\alpha}$. However, this contradicts with the fact that g can be applied to f_2, because the type $\langle\rangle$ in the type of $[\![f_2]\!]$ cannot contain any variable in $\overline{\alpha}_2$. Hence, we can conclude that A_1 cannot contain any variable in $\overline{\alpha}_1$, which will lead to contradiction when we consider the translation of f_1 $(\langle \ell = 1\rangle \triangleright \langle\rangle)$ because we can neither add the label ℓ in the type A_1, nor remove it in the type of $[\![\langle \ell = 1\rangle \triangleright \langle\rangle]\!]$.

□

E.2 Non-Existence of Type-Only Encodings of $\lambda^{\leqslant\text{co}}_{[]}$ in $\lambda^{\rho\theta}_{[]}$

Theorem 5.3. *There exists no global type-only encoding of $\lambda^{\leqslant\text{co}}_{[]}$ in $\lambda^{\rho\theta}_{[]}$.*

Proof. We assume that $\Delta = \alpha_0$ and $\Gamma = y : \alpha_0$ when environments are omitted. For simplicity, we omit the type of labels in variant types if it is α_0.

By the fact that $[\![-]\!]$ is type-only, we have:

- $(\ell\ y)^{[\ell]}$ is translated to $\Lambda\overline{\alpha}.(\ell\ (\Lambda\overline{\beta}.y))^{[R]}$ where $(\ell : \forall\overline{\beta}.\alpha_0) \in R$. By type preservation, we have $[\![[\ell]]\!] = \forall\overline{\alpha}.[R]$.
- $(\ell\ y)^{[\ell]} \triangleright [\ell;\ell']$ is translated to $\Lambda\overline{\tau}.[\![(\ell\ y)^{[\ell]}]\!]\ \overline{T} = \Lambda\overline{\tau}.(\Lambda\overline{\alpha}.(\ell\ (\Lambda\overline{\beta}.y))^{[R]})\ \overline{T}$ where $(\ell : \forall\overline{\beta}.\alpha_0) \in R$. By type preservation, we have $[\![[\ell;\ell']]\!] = (1)\forall\overline{\tau}\ \overline{\alpha}'_2.[R][\overline{T}/\overline{\alpha}'_1]$ where $\overline{\alpha} = \overline{\alpha}'_1\ \overline{\alpha}'_2$.
- $(\ell'\ y)^{[\ell;\ell']}$ is translated to $\Lambda\overline{\alpha}''.(\ell'\ (\Lambda\overline{\beta}''.y))^{[R'']}$ where $\ell' \in R''$. By symmetry, we also have $\ell \in R''$. By type preservation, we have $[\![[\ell;\ell']]\!] = (2)\forall\overline{\alpha}''.[R'']$.

By the fact that (1) = (2) and ℓ' can be an arbitrary label, we can conclude that R has a row variable ρ_R bound in $\overline{\alpha}'_1$ which is instantiated to the ℓ' label in R' by the substitution $[\overline{A}/\overline{\alpha}'_1]$. Thus, we have $(3) R = (\ell : \forall\overline{\beta}.\alpha_0);\ldots;\rho_R$ where $\rho_R \in \overline{\alpha}$.

Then, consider a nested variant $M = (\ell\ (\ell\ y)^{[\ell]})^{[\ell:[\ell]]}$. Because $[\![-]\!]$ is type-only, we have

$$[\![M]\!] = \Lambda\overline{\alpha}'.(\ell\ (\Lambda\overline{\beta}'.(\Lambda\overline{\alpha}.(\ell\ (\Lambda\overline{\beta}.y))^{[R]})\ \overline{A}))^{[R']}$$

By (3), $[\![M]\!]$ has type $\forall\overline{\alpha}'.[R'] = \forall\overline{\alpha}'.[(\ell : \forall\overline{\beta}'\ \overline{\alpha}_2.[R][\overline{A}/\overline{\alpha}_1]);\ldots]$, where $\overline{\alpha} = \overline{\alpha}_1\ \overline{\alpha}_2$ and $\rho_R \in \overline{\alpha}$.

We proceed by showing the contradiction that ρ_R can neither be in α_1 nor α_2.

$\rho_R \in \overline{\alpha}_2$ Consider $M' = (\ell\ (\ell\ y)^{[\ell;\ell']})^{[\ell:[\ell;\ell']]}$ of type $[\ell : [\ell;\ell']]$. By an analysis similar to M, it is easy to show that $[\![M']\!]$ has type $\forall\overline{\mu}.[(\ell : \forall\overline{\nu}.[R_1]);\ldots]$ where $\ell \in R_1$ and $\ell' \in R_1$.
Then, consider $M \triangleright [\ell : [\ell;\ell']]$ of the same type $[\ell : [\ell;\ell']]$ as M' which is translated to $\Lambda\overline{\gamma}.[\![M]\!]\ \overline{B}$. By type preservation, the translation of M' and $M \triangleright [\ell : [\ell;\ell']]$ should have the same type, which means R should contain label ℓ' after the type application of B. However, because $\rho_R \in \overline{\alpha}_2$, we cannot instantiate ρ_R to contain ℓ'. Besides, because ℓ' is arbitrarily chosen, it cannot already exist in R. Hence, $\rho_R \notin \overline{\alpha}_2$.

$\rho_R \in \overline{\alpha}_1$ Consider **case** $M\ \{\ell\ x \mapsto x \triangleright [\ell;\ell']\}$ of type $[\ell;\ell']$. By the type-only condition, it is translated to $(4)\Lambda\overline{\gamma}.\textbf{case}\ ([\![M]\!]\ \overline{C})\{\ell\ x \mapsto \Lambda\overline{\delta}.x\ \overline{D}\}$. By (2) we have $[\![[\ell;\ell']]\!] = \forall\overline{\alpha}''.[R'']$ where $\ell \in R''$ and $\ell' \in R''$. However, for (4), by the fact that $\rho_R \in \overline{\alpha}_1$ and $\overline{\alpha}_1$ are substituted by $\overline{A}$, the new row variable of the inner variant of M can only be bound in $\overline{\alpha}'$. Thus, in the case clause of ℓ, we cannot extend the variant type to contain ℓ' by type application of $\overline{D}$. Besides, because ℓ' is arbitrarily chosen and the translation is compositional, it can neither be already in the variant type or be introduced by the type application of $\overline{C}$. Hence, $\rho_R \notin \overline{\alpha}_1$.

Finally, by contradiction, the translation $[\![-]\!]$ does not exist.

□

E.3 Non-Existence of Type-Only Encodings of Full Subtyping

Theorem 5.4. *There exists no global type-only encoding of $\lambda^{\leqslant\text{full}}_{\langle\rangle}$ in $\lambda^{\rho\theta}_{\langle\rangle}$.*

Proof. Consider two functions $f_1 = \lambda x^{\langle\rangle}.x$ and $f_2 = \lambda x^{\langle\rangle}.\langle\rangle$ of the same type $\langle\rangle \to \langle\rangle$. By the type-only property, we have

$$\llbracket f_1 \rrbracket = \Lambda\overline{\alpha}_1.\lambda x^{A_1}.\Lambda\overline{\beta}_1.x\ \overline{B}_1$$
$$\llbracket f_2 \rrbracket = \Lambda\overline{\alpha}_2.\lambda x^{A_2}.\Lambda\overline{\beta}_2.\llbracket\langle\rangle\rrbracket\ \overline{B}_2 = \Lambda\overline{\alpha}_2.\lambda x^{A_2}.\Lambda\overline{\beta}_2.(\Lambda\overline{\gamma}.\langle\rangle)\ \overline{B}_2$$

By type preservation, they have the same type, which implies $x\ \overline{B}_1$ and $(\Lambda\overline{\gamma}.\langle\rangle)\ \overline{B}_2$ have the same type. We can further conclude that A_1 must be able to be instantiated to the empty record type $\langle\rangle$. Thus, the only way to have type variables bound by $\Lambda\overline{\alpha}_1$ in A_1 is to put them in the types of labels which are instantiated to be absent by the type application $x\ \overline{B}_1$.

Then, consider another two functions $g_1 = f_1 \rhd (\langle\ell : \langle\rangle\rangle \to \langle\rangle)$ and $g_2 = \lambda x^{\langle\ell:\langle\rangle\rangle}.(x.\ell)$ of the same type $\langle\ell : \langle\rangle\rangle \to \langle\rangle$. By the type-only property, we have

$$\llbracket g_1 \rrbracket = \Lambda\overline{\alpha}.\llbracket f_1 \rrbracket\ \overline{A} = \Lambda\overline{\alpha}.(\Lambda\overline{\alpha}_1.\lambda x^{A_1}.\Lambda\overline{\beta}_1.x\ \overline{B}_1)\ \overline{A}$$
$$\llbracket g_2 \rrbracket = \Lambda\overline{\alpha}'.\lambda x^{A'}.\Lambda\overline{\beta}'.\llbracket x.\ell \rrbracket\ \overline{B}' = \Lambda\overline{\alpha}'.\lambda x^{A'}.\Lambda\overline{\beta}'.(\Lambda\overline{\gamma}'.(x\ \overline{C}).\ell\ \overline{D})\ \overline{B}'$$

By type preservation, $\llbracket g_1 \rrbracket$ and $\llbracket g_2 \rrbracket$ have the same type. The $(x\ \overline{C}).l$ in $\llbracket g_2 \rrbracket$ implies that x has a polymorphic record type with label ℓ. Because ℓ is arbitrarily chosen, the only way to introduce ℓ in the parameter type of $\llbracket g_1 \rrbracket$ is by the type application of $\overline{A}$. However, we also have that type variables in $\overline{\alpha}_1$ can only appear in the types of labels in A_1, which means we cannot instantiate A_1 to be a polymorphic record type with the label ℓ by the type application of $\overline{A}$. Contradiction.

□