Gradual Typing for Extensibility by Rows

TARO SEKIYAMA, National Institute of Informatics and The Graduate University for Advanced Studies, SOKENDAI, Japan
ATSUSHI IGARASHI, Kyoto University, Japan

This work studies gradual typing for row types and row polymorphism. Key ingredients in this work are the *dynamic row type*, which represents a statically unknown part of a row, and *consistency for row types*, which allows injecting static row types into the dynamic row type and, conversely, projecting the dynamic row type to any static row type. While consistency captures the behavior of the dynamic row type statically, it makes the semantics of a gradually typed language incoherent when combined with row equivalence which identifies row types up to field reordering. To solve this problem, we develop *consistent equivalence*, which characterizes composition of consistency and row equivalence. Using consistent equivalence, we propose a polymorphic blame calculus $\mathrm{F}^{\rho}_{\mathrm{C}}$ for row types and row polymorphism. In $\mathrm{F}^{\rho}_{\mathrm{C}}$, casts perform not only run-time checking with the dynamic row type but also field reordering in row types. To simplify our technical development for row polymorphism, we adopt *scoped* labels, which are employed by the language Koka and are also emerging in the context of effect systems. We give the formal definition of $\mathrm{F}^{\rho}_{\mathrm{C}}$ with these technical developments and prove its type soundness. We also sketch the gradually typed surface language $\mathrm{F}^{\rho}_{\mathrm{G}}$ and type-preserving translation from $\mathrm{F}^{\rho}_{\mathrm{G}}$ to $\mathrm{F}^{\rho}_{\mathrm{C}}$ and discuss conservativity of $\mathrm{F}^{\rho}_{\mathrm{G}}$ over typing of a statically typed language with row types and row polymorphism.

Additional Key Words and Phrases: gradual typing, row types, row polymorphism

1 INTRODUCTION

1.1 Background: extensibility by rows and gradual typing

Extensibility is a measure of how open and how adaptive software is to future extensions or changes of system requirements. Extensibility is important not only for maintenance and adding new features but also for continuous, evolutionary software engineering practices such as Agile development. For example, consider a software system using a database. We may like to add a new column to a table in the database when extending the system, and to change the name of an existing column for refactoring. For such changes, software should be extensible—i.e., no modification to the existing code should be necessary except for the parts directly influenced by the changes.

One type-based approach to software extensibility is *row types* [Wand 1987], which address extensibility in terms of data types. A row type is a finite sequence $\ell_1 : A_1; \ldots; \ell_n : A_n$ of pairs of a label ℓ_i and its type A_i; it captures a common form of data types, such as record types and variant types, ubiquitous among various programming paradigms.

Row types are prominent in research on static typing and have been used in practice in many situations. From the outset, row types were developed for extensibility—Wand proposed row types for achieving extensibility originating from inheritance in object-oriented programming by records [Wand 1987, 1991]. That embedding of object-oriented features by row types is also adopted by the object system of OCaml in a more sophisticated way [Rémy and Vouillon 1998]. Row types are also able to make variant types extensible. Extensible variant types, also called polymorphic variants, are one of the techniques to resolve the Expression Problem [Garrigue 2000],

Authors' addresses: Taro Sekiyama, National Institute of Informatics and The Graduate University for Advanced Studies, SOKENDAI, Tokyo, Japan, tsekiyama@acm.org; Atsushi Igarashi, Kyoto University, Kyoto, Japan, igarashi@kuis.kyoto-u.ac.jp.

2019. XXXX-XXXX/2019/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

which is a litmus test to evaluate the suitability of a language for modular software development. Extensible variant types also provide a theoretical foundation for exceptions open to extension with user-defined errors. These extensible record and variant types with row types are implicitly or explicitly available in many languages such as OCaml, Haskell, PureScript, Gluon, Koka, etc. Another, more recent application of row types is to support effect systems for algebraic effects and handlers [Plotkin and Pretnar 2009], and multiple languages with such an effect system are emerging [Hillerström and Lindley 2016; Leijen 2014, 2017; Lindley et al. 2017].

While the practicality of row types has been demonstrated with many applications, strict enforcement of this static typing discipline might interfere with rapid software development, and in such contexts a dynamic typing discipline would be more suitable. On the other hand, as development progresses and software is scaled up, static typing provides more benefits, such as extensibility in a safe manner as well as early error detection and better maintainability.

Gradual typing, proposed independently by Siek and Taha [2006] and Tobin-Hochstadt and Felleisen [2006], has been studied for resolving the conflict between static and dynamic typing and for enabling gradual, smooth evolution from fully dynamically typed code to fully statically typed code. Gradual typing was first proposed for higher-order functions and later extended with various programming features such as subtyping [Siek and Taha 2007; Xie et al. 2018], parametric polymorphism [Ahmed et al. 2011, 2017; Igarashi et al. 2017; Toro et al. 2019; Xie et al. 2018], control operators [Sekiyama et al. 2015; Takikawa et al. 2013], and type inference [Garcia and Cimini 2015; Miyazaki et al. 2019; Siek and Vachharajani 2008]. A key ingredient for achieving gradual evolution is the *dynamic type*, denoted by $\star$, which is the type of dynamically typed code. The dynamic type makes it possible to inject any statically typed values into the dynamic type and, conversely, to project dynamically typed values to any static type with run-time type conversions, called *casts*. Gradual type systems reflect this semantic ability of the dynamic type to *consistency*. Consistency plays the role of type equality in gradual typing and tells where the cast is necessary. Consistency is designed to be flexible enough to allow possibly successful casts—e.g., between int and $\star$ and between int $\to \star$ and $\star \to$ bool—but strict enough not to miss definitely unsafe casts, e.g., between int and bool and between int $\to \star$ and bool $\to \star$.

1.2 Our work

This work aims at gradual evolution between dynamically typed code and statically typed, safely extensible code, and to this end we study gradual typing for row types. Key ingredients in our work are the *dynamic row type*, denoted by the same notation $\star$ as the dynamic type, and consistency for row types. The dynamic row type has been proposed first by Garcia et al. [2016] for making the effective use of monomorphic record types in gradual typing, and we extend it to handle variant types as well. The dynamic row type intuitively represents a statically unknown part of a row. For example, row type $\ell : \mathsf{int}; \star$ ensures that there is an ℓ field coupled with type int but it guarantees nothing about other fields, neither their presence nor absence. Thus, a record with that row type *must* have an ℓ field holding an integer value and *may* have other fields; a variant with that row type *requires* consumers of the variant to handle the case where the variant is constructed by injecting an integer value with label ℓ and *allows* them to handle other cases. We define consistency for row types taking into account this intuition. Interestingly, the dynamic row type not only enables gradual evolution of code with record and variant types but also provides fine-grained control over interfaces of program components, as seen in Section 2.

To bring extensibility achieved by static row typing into gradual typing, we also deal with *row polymorphism* [Gaster and Jones 1996; Wand 1987],[1] which gives great modularity and reusability

[1] Another major form of polymorphism is subtyping possibly with bounded polymorphism [Cardelli and Wegner 1985].

to components with row types by enabling a type signature of an expression to expose interesting fields and to abstract and take the remaining, uninteresting row information as a parameter. Introduction of row polymorphism to gradual typing, however, gives rise to two technical issues. The first is on row parametricity. To ensure row parametricity, we need to protect polymorphically typed values from untyped code. We resolve this issue by applying the idea in the earlier work on polymorphic gradual typing [Ahmed et al. 2011, 2017; Igarashi et al. 2017; Toro et al. 2019] to row polymorphism. The second issue is on row equivalence. In a monomorphic setting, we can assume that the label set is totally ordered and consider only the canonical form of a row. In a polymorphic setting, however, a row type obtained by substitution for a row type variable may not be in a canonical form, and therefore row types may be syntactically different even if they are semantically equivalent. A standard approach to this issue is to identify row types up to field reordering [Gaster and Jones 1996]. However, perhaps surprisingly, the semantics based on the earlier polymorphic gradual typing is *not* well defined in the sense that the behavior of some program changes depending on representative row types. To solve this problem, we develop *consistent equivalence*, which characterizes composition of consistency and row equivalence, and incorporate it into our gradually typed language instead of consistency and row equivalence. Thanks to consistent equivalence, the behavior of programs in the language—especially, the order of run-time checks—is determined by type annotations, not by representative row types. Thus, the behavior of a program is determined to be unique.

To ease technical development for row polymorphism, we allow duplicate labels in a single row; such labels are also called *scoped* [Leijen 2005]. An alternative approach to row polymorphism is to assume labels in a row to be unique and to introduce qualified types [Gaster and Jones 1996] or a kind system [Pottier and Rémy 2005] which assert that a row type variable can be instantiated only with rows without some labels. While we could give a gradually typed language with such a restriction on row variables, it would make the run-time checking of the language complicated. By contrast, row polymorphism with scoped labels does not need restriction on row variables and simplifies our technical development. In addition, the fact that emerging applications of row types—i.e., effect systems for algebraic effect handlers—adopt scoped labels [Biernacki et al. 2018; Leijen 2014, 2017; Lindley et al. 2017] motivates us to take this approach.

Employing scoped labels also enables us to use the *embedding operation* [Leijen 2005], which embeds a variant expression into a variant type with a wider row statically and wraps a variant value by a dummy label dynamically. The embedding operation was originally proposed to align rows in variant types with a polymorphic row variable, and its usefulness is also found in effect systems [Biernacki et al. 2018; Leijen 2014]. The embedding operation also plays an important role to make the type system of our calculus syntax-directed.

Below is a summary of the contributions by this work.

- We define consistency for value and row types. While consistency captures the essence of the dynamic type and the dynamic row type, it is problematic when used together with row equivalence. To solve the problem with consistency, we also give consistent equivalence.
- We define a polymorphic λ-calculus $\mathrm{F}^{\rho}_{\mathrm{C}}$ equipped with run-time checking by casts, row types with scoped labels, the dynamic row type, record and variant types, row polymorphism, and the embedding operation, using consistent equivalence.
- We show that consistent equivalence characterizes composition of consistency and row equivalence and that $\mathrm{F}^{\rho}_{\mathrm{C}}$ satisfies type soundness. We sketch a surface language $\mathrm{F}^{\rho}_{\mathrm{G}}$ for $\mathrm{F}^{\rho}_{\mathrm{C}}$ and type-preserving translation from $\mathrm{F}^{\rho}_{\mathrm{G}}$ to $\mathrm{F}^{\rho}_{\mathrm{C}}$ and also state conservativity of $\mathrm{F}^{\rho}_{\mathrm{G}}$ over typing of a statically typed language for row types and row polymorphism.

The rest of this paper is organized as follows. Section 2 presents motivating examples of records and variants with the dynamic row type. Next, we review a statically typed language F^{ρ} with row types and row polymorphism in Section 3. Section 4 defines consistency and consistent equivalence that support the dynamic row type and Section 5 formalizes $\mathrm{F}^{\rho}_{\mathrm{C}}$. Section 5 also sketches $\mathrm{F}^{\rho}_{\mathrm{G}}$ and translation from $\mathrm{F}^{\rho}_{\mathrm{G}}$ to $\mathrm{F}^{\rho}_{\mathrm{C}}$ and states properties of $\mathrm{F}^{\rho}_{\mathrm{C}}$ and $\mathrm{F}^{\rho}_{\mathrm{G}}$. After discussing related work in Section 6, we conclude in Section 7.

This paper omits some parts of definitions and the details of proofs. The full definitions, including those of $\mathrm{F}^{\rho}_{\mathrm{G}}$ and the translation from $\mathrm{F}^{\rho}_{\mathrm{G}}$ to $\mathrm{F}^{\rho}_{\mathrm{C}}$, and the complete proofs are found in the supplementary material.

2 PROGRAMMING WITH GRADUAL TYPING FOR ROW TYPES

Records and variants are fundamental building blocks to represent and manipulate data structures. Records provide a means to put several pieces of data together and to access them by specifying labels. Variants enables us to do case analysis on labels safely. This section shows multiple motivating examples of gradual typing for record and variant types. The programs presented in this section are in the surface language $\mathrm{F}^{\rho}_{\mathrm{G}}$, but it is easy to translate them to $\mathrm{F}^{\rho}_{\mathrm{C}}$.

2.1 Records

Gradual evolution of data structures. A trivial application of "gradualizing" record types is evolving shapes of data structures gradually. For instance, let us consider development of a window system. Assume that we have a function window that returns the current window frame.

When development starts with fully dynamic typing, window is given the dynamic type $\star$. Since a value of $\star$ can be supposed to have any type, we can use window as a function of unit $\to \star$. We also assume that a window frame is represented by a record. Then, for example, an expression checking that the current window is valid can be given as follows:

$$M \stackrel{\text{def}}{=} \mathsf{let}\ w = \mathsf{window}\,()\ \mathsf{in}\ w.\mathit{width} \le 2560\ \&\ w.\mathit{height} \le 1440.$$

which checks that the width and height of the current window frame are valid. Variable w bound to the current window is assigned type $\star$ and used as a record holding *width* and *height* fields having integer values. The static assumptions—whether window is a function and whether w is such a record—are checked at run time; for example, if the *width* field has a string value, then the run-time check for the *width* field will fail and an exception will be raised.

As development progresses, type specifications would gradually become concrete and stable. Now, suppose that the type of window is refined to be unit $\to [\mathit{width} : \mathsf{int}; \mathit{height} : \mathsf{int}; \star]$, where $\star$ is the *dynamic row type* and $[\rho]$ is a record type with row type ρ. Thus, this function type means that a window frame is represented by a record that holds *width* and *height* fields with integer values *surely* and, in addition, may hold other fields. Since this refinement is consistent with the assumptions on window and w in M, the expression M works still without any change. If the change is inconsistent with the assumption—e.g., the type of window is changed to unit $\to [\mathit{width} : \mathsf{str}; \mathit{height} : \mathsf{str}; \star]$—the type system would detect the type mismatch statically.

The dynamic row type $\star$ left in the record type indicates a possibility that a window frame has other field specifications which are not fixed. This gives the ability to develop a prototype implementation rapidly. For example, let us consider prototype development of window drawing in the stack order, where a window frame with lower *depth* field is drawn in front of other windows with greater *depth* fields. Since window returns the current window frame, it should be the topmost, i.e., its *depth* field should be 0. Thus, the checking expression would be rewritten as:

$$\mathsf{let}\ w = \mathsf{window}\,()\ \mathsf{in}\ w.\mathit{width} \le 2560\ \&\ w.\mathit{height} \le 1440\ \&\ w.\mathit{depth} = 0.$$

Here, we do not need to change the type of window because the dynamic row type allows us to *suppose* the window frame to have a *depth* field. This flexibility of the dynamic row type lets us concentrate on extending software and avoid being bothered by type puzzles. Once it is decided to deploy this drawing system into production, we could opt to detect typing errors statically and make the software safer by changing the record type to $[\mathit{width} : \mathsf{int}; \mathit{height} : \mathsf{int}; \mathit{depth} : \mathsf{int}; \star]$.

Optional information. Record types combined with the dynamic row type are also useful to attach optional information. For example, let us consider a function that tests if a given string matches a given regular expression and returns not only the testing result of Boolean but also a matching substring if the test succeeds. We also suppose that users have to give an option in order to make the function return the matching substring for reducing memory consumption. We can give such a function matching the following type:

$$\mathsf{val}\ \ \mathsf{matching}\ \ :\ \ [\mathit{re} : \mathsf{str}; \mathit{match} : \mathsf{str}; \star] \to [\mathit{res} : \mathsf{bool}; \star].$$

The fields that appear explicitly in the argument type are *mandatory* arguments: users have to give a regular expression by the *re* field and a string to match by the *match* field. The dynamic row type there corresponds to *optional* arguments: in order for the function to return a matching substring, one sets the *return_sub* field to true:[2]

$$M \stackrel{\text{def}}{=} \mathsf{matching}\,\{\mathit{re} = \texttt{"o}^*\texttt{"}; \mathit{match} = \texttt{"foo"}; \mathit{return\_sub} = \mathsf{true}\}.$$

The return type of matching means that matching returns whether the string matches the pattern by the Boolean *res* field. The dynamic row type in the return type enables augmenting the Boolean result with the matching substring, if any, by the *substr* field. Then, we can write a program that returns the length of the matched substring (if any) or returns -1.

$$\mathsf{let}\ x : [\mathit{res} : \mathsf{bool}; \star] = M\ \mathsf{in}\ \mathsf{if}\ x.\mathit{res}\ \mathsf{then}\ (\mathsf{length}\ x.\mathit{substr})\ \mathsf{else}\ -1$$

matching does not produce the matching substring if the *return_sub* field is missing or set to false:

$$\mathsf{let}\ x : [\mathit{res} : \mathsf{bool}; \star] = \mathsf{matching}\,\{\mathit{re} = \texttt{"o}^*\texttt{"}; \mathit{match} = \texttt{"foo"}\}\ \mathsf{in}\ x.\mathit{substr} \longrightarrow^* \mathit{exception}$$

Thus, the dynamic row type can give natural and flexible type interfaces beyond gradual evolution.

Dynamic data type definition. The dynamic row type in record types is also useful when one deals with values whose structures are determined by external environments. For example, loading JSON files and constructing object-relational mappings by analyzing SQL queries at run time are such practical applications.

2.2 Variants

A key operation on variants is injection, which injects values of different types into a single type representation, a variant type $\langle\rho\rangle$, by tagging the values with labels that occur in row type ρ. The injected values can be projected to the field types of ρ safely. Variants are seen throughout programming–their applications include enumerated types, heterogeneous collections, and algebraic data types, sometimes together with recursive types.

Variant types combined with the dynamic row type not only allow gradual evolution of code with variant types but also can represent cases with uncertainty. For example, let us consider a function input_event that returns an input event from users. There are several kinds of events,

[2]Here we assume that a language supports *dynamic field testing* on records. We do not deal with such an operation in this paper, but it is easy to add, like type testing on dynamically typed values [Ahmed et al. 2011].

such as key down, key up, mouse move, mouse click, etc. We can use variant types to represent what event happens with additional information of the event (e.g., key codes if key events happen).

$$\mathsf{val\ input\_event\ :\ unit} \to \langle \mathit{key\_down} : \mathsf{int}; \mathit{key\_up} : \mathsf{int}; \ldots; \cdot \rangle$$

where $\cdot$ is the empty row. Suppose that we have to handle all key and mouse events but do not have to handle events from other input devices such as touchscreens and gamepads. We could naturally imagine that input_event is changed to take optional arguments to specify what additional events we are interested in:

$$\mathsf{val\ input\_event\ :\ [\star]} \to \langle \mathit{key\_down} : \mathsf{int}; \mathit{key\_up} : \mathsf{int}; \ldots; \cdot \rangle.$$

For example, if we are interested in touchscreen events as well, we would call input_event with an additional argument to enable monitoring touchscreen events, like:

$$\mathsf{input\_event}\,\{\mathit{touch} = \mathsf{true}\}.$$

For the return type of input_event, enumerating all possible events in the variant type would be inconvenient from the viewpoints of both efficiency and engineering because it seems that we have to handle even uninteresting, not happening events. Variant types with the dynamic row type allow us to take care of only mandatory and interesting events by changing the type signature of input_event as:

$$\mathsf{val\ input\_event\ :\ [\star]} \to \langle \mathit{key\_down} : \mathsf{int}; \mathit{key\_up} : \mathsf{int}; \ldots; \star \rangle$$

where *key_down*, *key_up*, ... are mandatory events that must be handled and $\star$ in the return type is for events handled only when interesting. If we do not have additional interesting events, we can convert $\star$ to the empty row $\cdot$:

$$\mathsf{input\_event}\,\{\} : \langle \mathit{key\_down} : \mathsf{int}; \mathit{key\_up} : \mathsf{int}; \ldots; \cdot \rangle.$$

If interested in touchscreen devices, we can convert $\star$ to fields for touchscreen events:

$$\mathsf{input\_event}\,\{\mathit{touch} = \mathsf{true}\} : \langle \mathit{key\_up} : \mathsf{int}; \ldots; \mathit{touch\_start} : \mathsf{pos}; \mathit{touch\_end} : \mathsf{pos}; \cdot \rangle$$

where pos is the type of positions. While we can choose optional events by passing an optional argument and converting $\star$, we *cannot* drop mandatory events, such as *key_up* and *key_down*.

Furthermore, the flexibility of the dynamic row type makes it possible to monitor events even from devices unknown to the provider of input_event. Let us suppose that input_event supports dynamic loading of device driver libraries to monitor events from unknown devices. Such events could not appear in a type signature of input_event because input_event does not know at compile time what events will be triggered by an unknown device, though it can know at run time by dynamic library loading. The dynamic row type enables users of input_event to assert what events are monitored by input_event when a device driver is loaded. For example, if a barcode reader is not supported by input_event but it provides a device driver library, we can assert that an event from the barcode reader may happen by converting $\star$:

$$\mathsf{input\_event}\,\{\mathit{load} = \texttt{"barcode\_lib"}\} : \langle \mathit{key\_up} : \mathsf{int}; \ldots; \mathit{barcode} : \mathsf{str}; \cdot \rangle.$$

It would be difficult to give this flexibility only by static typing.

3 A POLYMORPHICALLY TYPED LANGUAGE FOR ROW TYPES

We start with reviewing a statically typed language F^{ρ} with row types and row polymorphism. Our language F^{ρ} is a variant of the language given by Hillerström et al. [2017], from which F^{ρ} differs in that it adopts scoped labels and incorporates row equivalence as a typing rule.

$$
\begin{array}{lrcl}
\textbf{Variables for types and rows} & X & \textbf{Kinds} & K ::= \mathsf{T} \mid \mathsf{R} \\
\textbf{Base types} & \iota ::= \mathsf{bool} \mid \mathsf{int} \mid \ldots & \textbf{Constants} & \kappa ::= \mathsf{true} \mid \mathsf{false} \mid 0 \mid + \mid \ldots \\
\textbf{Types and rows} & A, B, C, D, \rho & ::= & X \mid \iota \mid A \to B \mid \forall X{:}K.\, A \mid [\rho] \mid \langle \rho \rangle \mid \cdot \mid \ell : A; \rho \\
\textbf{Terms} & M & ::= & x \mid \kappa \mid \lambda x{:}A.M \mid M_1\, M_2 \mid \Lambda X{:}K.\, M \mid M\, A \mid \\
 & & & \{\} \mid \{\ell = M_1; M_2\} \mid \mathsf{let}\, \{\ell = x; y\} = M_1\, \mathsf{in}\, M_2 \mid \\
 & & & \ell\, M \mid \mathsf{case}\, M\, \mathsf{with}\, \langle \ell\, x \to M_1; y \to M_2 \rangle \mid \uparrow \langle \ell : A \rangle\, M \\
\multicolumn{4}{l}{\textbf{Values} \quad w ::= \kappa \mid \lambda x{:}A.M \mid \Lambda X{:}K.\, M \mid \{\} \mid \{\ell = w_1; w_2\} \mid w^{\ell} \qquad w^{\ell} ::= \ell\, w \mid \uparrow \langle \ell : A \rangle\, w^{\ell}} \\
\textbf{Evaluation contexts} & F & ::= & [\,] \mid F\, M_2 \mid w_1\, F \mid F\, A \mid \\
 & & & \{\ell = F; M_2\} \mid \{\ell = w_1; F\} \mid \mathsf{let}\, \{\ell = x; y\} = F\, \mathsf{in}\, M_2 \mid \\
 & & & \ell\, F \mid \mathsf{case}\, F\, \mathsf{with}\, \langle \ell\, x \to M_1; y \to M_2 \rangle \mid \uparrow \langle \ell : A \rangle\, F \\
\textbf{Typing contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x{:}A \mid \Gamma, X{:}K
\end{array}
$$

Fig. 1. Syntax of F^{ρ}.

3.1 Syntax

Figure 1 defines the syntax of F^{ρ}, a statically typed λ-calculus equipped with polymorphism, records, variants, and a kind system to classify value types and row types. Metavariable X ranges over type and row variables and K over kinds. Kind T is the kind of value types, and R is that of row types. We often just say "types" for value types and "rows" for row types. Evaluation contexts F and typing contexts Γ are defined in a standard manner.

Types and rows. We use A, B, C, and D to mean types and ρ to mean rows. Types are: variables X; base types ι; function types $A \to B$; universal types $\forall X{:}K.\, A$, where X is bound in A and it will be instantiated with inhabitants of K; record types $[\rho]$; or variant types $\langle \rho \rangle$. Rows are variables, the empty row $\cdot$, or extension $(\ell : A; \rho)$ of row ρ with label ℓ and A. For example, $(\ell_1 : \mathsf{int}; \ell_2 : \mathsf{bool}; \cdot)$ is a row type having two fields, ℓ_1 with int and ℓ_2 with bool. Record type $[\ell_1 : \mathsf{int}; \ell_2 : \mathsf{bool}; \cdot]$ is given to records that hold an integer value accessed by ℓ_1 and a Boolean value accessed by ℓ_2. Variant type $\langle \ell_1 : \mathsf{int}; \ell_2 : \mathsf{bool}; \cdot \rangle$ is given to an integer value tagged with ℓ_1 or a Boolean value tagged with ℓ_2. Types and rows are not distinguished by the syntax; they are by the kind system given in Section 3.3.

We make a remark on scoped (i.e., duplicate) labels. For example, scoped labels allow row type $(\ell : \mathsf{int}; \ell : \mathsf{bool}; \cdot)$ though the same label ℓ occurs twice there. Scoped labels make row polymorphism easy to use. For example, let us consider a function that removes field $\ell_1 : A$ from a given record and instead appends field $\ell_2 : B$ to it. A promising type of that function would be $\forall X{:}\mathsf{R}.\, [\ell_1 : A; X] \to [\ell_2 : B; X]$, and, indeed, F^{ρ} would allow it to have that type. Similarly, a function that handles only the case that a given variant is tagged with label ℓ_1 would be able to have type $\forall X{:}\mathsf{R}.\, \langle \ell_1 : A; X \rangle \to B$ for some type B. These type representations are acceptable thanks to scoped labels. In other words, if labels in F^{ρ} were not scoped (i.e., had to be unique in a row), F^{ρ} would not allow such type representations because row variable X may be instantiated with a row including a field with label ℓ_1 or ℓ_2.

Terms and values. Terms are ranged over by M. In Figure 1, the first line for terms—i.e., variables x; constants κ; functions $\lambda x{:}A.M$, where x is bound in M; function applications $M_1\, M_2$; type abstractions $\Lambda X{:}K.\, M$, where X is bound in M; and type applications $M\, A$—comes from System F [Reynolds 1974]. The only difference from System F is that type abstractions abstract not only over value types but also over row types.

The second line shows operations on records: $\{\}$ is the empty record; $\{\ell = M_1; M_2\}$ is the extension of record M_2 with M using label ℓ; and record decomposition $\mathsf{let}\ \{\ell = x; y\} = M_1\ \mathsf{in}\ M_2$ decomposes the record of M_1 into a value held by the outermost ℓ field and the rest of the record and binds x to the value and y to the remaining record in M_2. These operations are fundamental enough to implement the basic operations on records [Cardelli and Mitchell 1991; Leijen 2005]: Extension just corresponds to the record extension $\{\ell = M_1; M_2\}$; Restriction, which removes an ℓ field from a record M, is implemented by $\mathsf{let}\ \{\ell = x; y\} = M\ \mathsf{in}\ y$; Extraction (written $M.\ell$ in Section 2), which extracts the value of an ℓ field from a record M, is by $\mathsf{let}\ \{\ell = x; y\} = M\ \mathsf{in}\ x$. The notation $\{\ell_1 = M_1; \ldots; \ell_n = M_n\}$ used in Section 2 is an abbreviation of $\{\ell_1 = M_1; \{\ldots; \{\ell_n = M_n; \{\}\} \ldots\}\}$.

Terms in the third line are for variants. Injection $(\ell\ M)$ tags the value of M with ℓ. Case expression $\mathsf{case}\ M\ \mathsf{with}\ \langle \ell\ x \to M_1; y \to M_2\rangle$ (where x and y are bound in M_1 and M_2 respectively) tests the variant value of M on ℓ; if it is tagged with ℓ, M_1 will be evaluated with binding of x to the injected value; otherwise, M_2 will be evaluated with binding of y to the variant. The last is the so-called *embedding operation* [Leijen 2005], tailored to variant types with scoped labels. Embedding $\uparrow\langle \ell : A\rangle\ M$ embeds the variant value of M into a variant type extended with field $\ell : A$. That is, if M has type $\langle \rho\rangle$, the type of embedding term $\uparrow\langle \ell : A\rangle\ M$ is $\langle \ell : A; \rho\rangle$. The embedding operation may seem to be just an operation to enable width subtyping. This is the case if it is sure that ρ never contains any ℓ field. However, if ρ could contain an ℓ field—this includes the case that ρ ends with a row variable because it may be instantiated with a row holding an ℓ field—it is not the case. In such a case, the embedding operation works as inserting a dummy field with label ℓ, and the label ℓ attached by the embedding operation does not match with the label ℓ in a case expression. Instead, the case expression peels off the label given by the embedding operation. For instance, case expression $\mathsf{case}\ \uparrow\langle \ell : A\rangle\ (\ell\ M)\ \mathsf{with}\ \langle \ell\ x \to M_1; y \to M_2\rangle$ will be reduced to M_2 with binding of y to the value of $(\ell\ M)$. The embedding operation is useful especially to align variant types containing row variables. For example, suppose that an expression M has type $\langle X\rangle$ and consider writing a program that returns M if some condition M' holds and, otherwise, returns $\ell\ 0$. We can make such a program acceptable using the embedding operation:

$$\mathsf{if}\ M'\ \mathsf{then}\ \uparrow\langle \ell : \mathsf{int}\rangle\ M\ \mathsf{else}\ (\ell\ 0).$$

Without the embedding operation, the program would be rejected because terms of $\langle X\rangle$ could not have any variant type including an ℓ field. More practical applications of the embedding operation can be found in the literature on effects [Biernacki et al. 2018; Leijen 2014].

Values, ranged over w, are constants, functions, type abstractions, the empty row, records holding only values, or variant values. Variant values, ranged over by w^ℓ, are values injected with label ℓ or application of the embedding operation to a variant value with ℓ. Note that injection and embedding in a variant value w^ℓ shares the same label ℓ.

Notation. We introduce standard notions and notation. The set of type and row variables that occur free in A is written $ftv(A)$. We define capture-avoiding substitution $M\,[w/x]$ (resp. $M\,[A/X]$) of w (resp. A) for x (resp. X) in M as usual. We also write $A[B/X]$ for the capture avoiding substitution of B for X in A. Filling the hole of evaluation context F with term M is denoted by $F[M]$. We write $dom\,(\Gamma)$ for the set of variables (both x and X) bound by Γ. We use similar notation for other syntax classes throughout the paper.

3.2 Semantics

The semantics of F^ρ is given by two relations between terms: the reduction rule $\rightsquigarrow^s$ and the evaluation relation $\longrightarrow^s$, which are defined by the rules in Figure 2.

Reduction rules $\boxed{M_1 \rightsquigarrow^s M_2}$ **Evaluation rule** $\boxed{M_1 \longrightarrow^s M_2}$

$$\kappa_1\,\kappa_2 \rightsquigarrow^s \zeta(\kappa_1,\kappa_2) \quad \text{Rs\_Const} \qquad (\lambda x{:}A.M)\,w \rightsquigarrow^s M[w/x] \quad \text{Rs\_Beta} \qquad (\Lambda X{:}K.\,M)\,A \rightsquigarrow^s M[A/X] \quad \text{Rs\_TyBeta}$$

$$\text{let}\,\{\ell = x; y\} = w\,\text{in}\,M_2 \rightsquigarrow^s M_2[w_1/x, w_2/y] \quad (\text{if } w \triangleright_\ell w_1, w_2) \quad \text{Rs\_Record}$$

$$\uparrow\langle \ell : A\rangle\,(w^{\ell'}) \rightsquigarrow^s w^{\ell'} \quad (\text{if } \ell \neq \ell') \quad \text{Rs\_Embed}$$

$$\text{case}\,(\ell\,w)\,\text{with}\,\langle \ell\,x \to M_1; y \to M_2\rangle \rightsquigarrow^s M_1[w/x] \quad \text{Rs\_CaseL}$$

$$\text{case}\,\uparrow\langle \ell : A\rangle\,(w^{\ell})\,\text{with}\,\langle \ell\,x \to M_1; y \to M_2\rangle \rightsquigarrow^s M_2[w^{\ell}/y] \quad \text{Rs\_CaseR1}$$

$$\text{case}\,w^{\ell'}\,\text{with}\,\langle \ell\,x \to M_1; y \to M_2\rangle \rightsquigarrow^s M_2[w^{\ell'}/y] \quad (\text{if } \ell \neq \ell') \quad \text{Rs\_CaseR2}$$

$$F[M_1] \longrightarrow^s F[M_2] \quad (\text{if } M_1 \rightsquigarrow^s M_2) \quad \text{Es\_Red}$$

Fig. 2. Semantics of F^ρ.

The reduction rules are shown at the top of Figure 2. The first three rules are standard. Reduction of constant application $\kappa_1\,\kappa_2$ depends on the denotation mapping ζ, which maps a pair of constants to the constant corresponding to their denotation. Function and type applications reduce to the bodies of the abstractions with substitution of the arguments. The rule (Rs_Record) splits a record value w into w_1, which is associated to ℓ, and w_2, which is the result of removing w_1 from record w. The values w_1 and w_2 are obtained by splitting function $\triangleright_\ell$ defined as follows.

Definition 1 (Record splitting). *$w \triangleright_\ell w_1, w_2$ is defined as follows:*

$$\{\ell = w_1; w_2\} \triangleright_\ell w_1, w_2 \qquad \{\ell' = w_1; w_2\} \triangleright_\ell w_{21}, \{\ell' = w_1; w_{22}\}\ (\text{if } \ell \neq \ell' \text{ and } w_2 \triangleright_\ell w_{21}, w_{22})$$

Then, the subsequent term M_2 will be executed after substituting w_1 and w_2.

The last four reduction rules are for variants. The first is for embedding terms, and it means that embedding is discarded if a label of an embedding term is different from the one of the variant value w^ℓ. This is justified by the fact that a variant type of w^ℓ can contain any field other than ℓ fields and, therefore, only retaining applications of the embedding operation with the same label ℓ is important. The other rules are for case expressions $\text{case}\,w^{\ell'}\,\text{with}\,\langle \ell\,x \to M_1; y \to M_2\rangle$. If $w^{\ell'}$ is an injection with ℓ, the branch M_1 will be evaluated with substitution of the injected value for x (Rs_CaseL). If $w^{\ell'}$ is an embedding term with ℓ, as explained above, M_2 will be evaluated with substitution of the underlying variant value for y (Rs_CaseR1). If $\ell \neq \ell'$, M_2 will be evaluated with substitution of the same variant value for y (Rs_CaseR2).

3.3 Type system

As other type systems for row types, the type system of F^ρ also identifies row types up to reordering of fields only with *distinct* labels [Berthomieu and le Moniès de Sagazan 1995; Leijen 2005].

Definition 2 (Type-and-row equivalence). *Type-and-row equivalence $A \equiv B$ is the smallest congruence relation satisfying the following rule:*

$$\frac{\ell \neq \ell'}{\ell : A; \ell' : B; \rho \equiv \ell' : B; \ell : A; \rho}\ \textit{Eq\_Swap}$$

For example, this definition deems row type $(\ell_1 : \text{int}; \ell_2 : \text{bool}; \rho)$ equivalent to $(\ell_2 : \text{bool}; \ell_1 : \text{int}; \rho)$ if and only if $\ell_1 \neq \ell_2$. The restriction on inequality of labels is necessary for type soundness. For example, a record $\{\ell = 0; \{\ell = \text{true}; \{\}\}\}$ should not be typed at $[\ell : \text{bool}; \ell : \text{int}; \cdot]$ because record decomposition for ℓ extracts the value of the outermost ℓ field.

The type system of F^ρ is given by three judgments: well-formedness of typing contexts $\vdash^s \Gamma$, well-formedness of types $\Gamma \vdash^s A : K$, and typing judgment $\Gamma \vdash^s M : A$. The inference rules of these judgments are in Figure 3 (where trivial well-formedness rules are omitted), and most of

Well-formedness rules (selected) $\boxed{\vdash^s \Gamma}$ $\boxed{\Gamma \vdash^s A : K}$

$$\frac{\vdash^s \Gamma \quad X{:}K \in \Gamma}{\Gamma \vdash^s X : K}\ \textsc{WFs\_TyVar} \qquad \frac{\Gamma, X{:}K \vdash^s A : \mathsf{T}}{\Gamma \vdash^s \forall X{:}K.\, A : \mathsf{T}}\ \textsc{WFs\_Poly} \qquad \frac{\Gamma \vdash^s \rho : \mathsf{R}}{\Gamma \vdash^s [\rho] : \mathsf{T}}\ \textsc{WFs\_Record}$$

$$\frac{\Gamma \vdash^s \rho : \mathsf{R}}{\Gamma \vdash^s \langle \rho \rangle : \mathsf{T}}\ \textsc{WFs\_Variant} \qquad \frac{\vdash^s \Gamma}{\Gamma \vdash^s \cdot : \mathsf{R}}\ \textsc{WFs\_REmp} \qquad \frac{\Gamma \vdash^s A : \mathsf{T} \quad \Gamma \vdash^s \rho : \mathsf{R}}{\Gamma \vdash^s \ell : A; \rho : \mathsf{R}}\ \textsc{WFs\_Cons}$$

Typing rules $\boxed{\Gamma \vdash^s M : A}$

$$\frac{\vdash^s \Gamma \quad x{:}A \in \Gamma}{\Gamma \vdash^s x : A}\ \textsc{Ts\_Var} \qquad \frac{\vdash^s \Gamma}{\Gamma \vdash^s \kappa : ty(\kappa)}\ \textsc{Ts\_Const} \qquad \frac{\Gamma, x{:}A \vdash^s M : B}{\Gamma \vdash^s \lambda x{:}A.M : A \to B}\ \textsc{Ts\_Lam}$$

$$\frac{\Gamma \vdash^s M_1 : A \to B \quad \Gamma \vdash^s M_2 : A}{\Gamma \vdash^s M_1\, M_2 : B}\ \textsc{Ts\_App} \qquad \frac{\Gamma, X{:}K \vdash^s M : A}{\Gamma \vdash^s \Lambda X{:}K.\, M : \forall X{:}K.\, A}\ \textsc{Ts\_TLam}$$

$$\frac{\Gamma \vdash^s M : \forall X{:}K.\, A \quad \Gamma \vdash^s B : K}{\Gamma \vdash^s M\, B : A[B/X]}\ \textsc{Ts\_TApp} \qquad \frac{\vdash^s \Gamma}{\Gamma \vdash^s \{\} : [\cdot]}\ \textsc{Ts\_REmp}$$

$$\frac{\Gamma \vdash^s M_1 : A \quad \Gamma \vdash^s M_2 : [\rho]}{\Gamma \vdash^s \{\ell = M_1; M_2\} : [\ell : A; \rho]}\ \textsc{Ts\_RExt} \qquad \frac{\Gamma \vdash^s M_1 : [\ell : A; \rho] \quad \Gamma, x{:}A, y{:}[\rho] \vdash^s M_2 : B}{\Gamma \vdash^s \mathsf{let}\ \{\ell = x; y\} = M_1\ \mathsf{in}\ M_2 : B}\ \textsc{Ts\_RLet}$$

$$\frac{\Gamma \vdash^s M : A \quad \Gamma \vdash^s \rho : \mathsf{R}}{\Gamma \vdash^s \ell\, M : \langle \ell : A; \rho \rangle}\ \textsc{Ts\_VInj} \qquad \frac{\Gamma \vdash^s M : \langle \rho \rangle \quad \Gamma \vdash^s A : \mathsf{T}}{\Gamma \vdash^s \uparrow \langle \ell : A \rangle\, M : \langle \ell : A; \rho \rangle}\ \textsc{Ts\_VLift}$$

$$\frac{\Gamma \vdash^s M : \langle \ell : A; \rho \rangle \quad \Gamma, x{:}A \vdash^s M_1 : B \quad \Gamma, y{:}\langle \rho \rangle \vdash^s M_2 : B}{\Gamma \vdash^s \mathsf{case}\ M\ \mathsf{with}\ \langle \ell\, x \to M_1; y \to M_2 \rangle : B}\ \textsc{Ts\_VCase}$$

$$\frac{\Gamma \vdash^s M : A \quad A \equiv B \quad \Gamma \vdash^s B : \mathsf{T}}{\Gamma \vdash^s M : B}\ \textsc{Ts\_Equiv}$$

Fig. 3. The type system of F^{ρ}.

them are standard or easy to understand. We explain only the key rules in what follows. The rules for well-formedness of types assign kind T to value types and R to row types; the kind of a type variable is given by a typing context (WFs_TyVar). The type of a constant κ is assigned by function ty (Ts_Const); we assume that the type respects the denotation of κ. Injection $\ell\, M$ can be given any variant type where the first ℓ field has the same type as M. Embedding $\uparrow \langle \ell : A \rangle\, M$ extends the variant type of M with field $\ell : A$. For case expression $\mathsf{case}\, M\, \mathsf{with}\, \langle \ell\, x \to M_1; y \to M_2 \rangle$, matched expression M must have a variant type holding an ℓ field and branches M_1 and M_2 are typechecked under the assumptions that x and y are bound to a value injected with ℓ and a variant value discarding the first ℓ field, respectively. The last rule (Ts_Equiv) allows reordering of fields with distinct labels by employing type-and-row equivalence. Thanks to (Ts_Equiv), the type system can accept terms like:

$$\lambda f{:}\forall X{:}\mathsf{R}.\, [\ell_1 : \mathsf{int}; X] \to A.f\, (\ell_2 : \mathsf{bool}; \cdot)\, \{\ell_2 = \mathsf{true}; \{\ell_1 = 0; \{\}\}\}.$$

This term would be rejected without (Ts_Equiv), because $f\, (\ell_2 : \mathsf{bool}; \cdot)$ requires arguments of $[\ell_1 : \mathsf{int}; \ell_2 : \mathsf{bool}; \cdot]$ but the type of the actual argument $\{\ell_2 = \mathsf{true}; \{\ell_1 = 0; \{\}\}\}$ is $[\ell_2 : \mathsf{bool}; \ell_1 : \mathsf{int}; \cdot]$, which is syntactically different from the type required by $f\, (\ell_2 : \mathsf{bool}; \cdot)$. Type-and-row equivalence makes these two record types interchangeable and, therefore, the above function application is accepted by giving $[\ell_1 : \mathsf{int}; \ell_2 : \mathsf{bool}; \cdot]$ to the argument record.

4 CONSISTENCY AND CONSISTENT EQUIVALENCE

This section presents *consistency*. Consistency for row types allows the dynamic row type to be interpreted as any row. However, consistency does not consider type-and-row equivalence, which is problematic when we derive a gradually typed language from F^ρ with implicit type conversion by type-and-row equivalence. To resolve the issue on consistency, we introduce *consistent equivalence*, which characterizes composition of type-and-row equivalence and consistency.

In this section, we consider *gradual* types and rows, which are obtained by extending static types given in Figure 1 with $\star$, which denote the dynamic type or the dynamic row type depending on contexts.

$$A, B, C, D, \rho \quad ::= \quad X \mid \star \mid \iota \mid A \to B \mid \forall X{:}K.\, A \mid [\rho] \mid \langle \rho \rangle \mid \cdot \mid \ell : A; \rho$$

We show the kind system for the extended types in Section 5.

4.1 Consistency

Consistency $\sim$ is fundamental to the static aspect of gradual typing and decides possible interaction between statically typed and dynamically typed code. Usually, it is defined as a binary relation between types and, intuitively, types are consistent if casts between them could be successful. For example, statically typed values can be injected to the dynamic type and, conversely, dynamically typed values could be projected to any type (whether a cast succeeds depends on whether run-time values can behave as the target type of the cast, though). This is axiomatized by the following rules.

$$A \sim \star \qquad \star \sim A$$

Consistency is also defined so that type constructors are compatible with it. For example, a consistency rule for function types is:

$$\frac{A_1 \sim B_1 \quad A_2 \sim B_2}{A_1 \to A_2 \sim B_1 \to B_2}$$

In what follows, we discuss how to extend consistency to deal with row types and universal types and then give its formal definition. After that, we show issues with consistency in designing a gradually typed language with it. These issues motivate us to introduce consistent equivalence.

4.1.1 Consistency for row types. A trivial extension of consistency to row types is to allow relating the dynamic row type to any row type ($\rho \sim \star$ and $\star \sim \rho$) and to add the following compatible rules for the empty row and row extension.

$$\cdot \sim \cdot \qquad \frac{A \sim B \quad \rho_1 \sim \rho_2}{\ell : A; \rho_1 \sim \ell : B; \rho_2}$$

These rules make, e.g., $(\ell : \mathsf{int}; \cdot)$ and $(\ell : \star; \cdot)$ consistent.

While necessary and reasonable, these compatibility rules are not sufficient to contain all pairs of row types such that casts between them could be successful. The problem is in a case that row types to be related end with $\star$ (i.e., they take the form $\ell_1 : A_1; \ldots; \ell_n : A_n; \star$) and they hold field labels distinct from those of each other. For example, let us consider row types $\ell_1 : \mathsf{int}; \star$ and $\ell_2 : \mathsf{str}; \star$ where $\ell_1 \neq \ell_2$. While these row types are not consistent only with the above extension of consistency, it is desirable that they are consistent because casts between record types and between variant types with these rows could be successful. Casts between record types $[\ell_1 : \mathsf{int}; \star]$ and $[\ell_2 : \mathsf{str}; \star]$ could be successful because a record value of either of them could hold both ℓ_1 and ℓ_2 fields. Similarly, casts between variant types $\langle \ell_1 : \mathsf{int}; \star \rangle$ and $\langle \ell_2 : \mathsf{str}; \star \rangle$ could be successful because they accommodate both of values injected with ℓ_1 and ℓ_2. It is notable that the assumption that ℓ_1 and ℓ_2 are distinct labels is critical here. For example, $[\ell_1 : \mathsf{int}; \star]$ and $[\ell_1 : \mathsf{str}; \star]$ should not be consistent since the types int and str of their ℓ fields are inconsistent.

The example showing the insufficiency of the simple extension above guides the design of a new consistency rule for row extension: given consistent row types ρ_1 and ρ_2, extension of ρ_1 with label ℓ preserves consistency with ρ_2 if ρ_2 ends with $\star$ and ℓ does not appear in ρ_2 . Formally:

$$\frac{\ell \notin \mathit{dom}(\rho_2) \quad \rho_2 \text{ ends with } \star \quad \rho_1 \sim \rho_2}{\ell : A; \rho_1 \sim \rho_2}\ \text{C\_ConsL}$$

where $\mathit{dom}(\rho_2)$ is the set of the field labels of ρ_2. This rule is justified by the intuition that: first, the occurrence of $\star$ in ρ_2 allows assuming that ρ_2 could contain a field of $\ell : \star$; and then the ℓ field can move to the head of ρ_2 by type-and-row equivalence since ρ_2 is assumed not to have other ℓ fields. We can apply the same discussion for extension of ρ_2 and indeed require consistency to satisfy the symmetric version of (C_ConsL). Then, row types $(\ell_1 : \mathsf{int}; \star)$ and $(\ell_2 : \mathsf{str}; \star)$ are consistent.

4.1.2 Consistency for universal types. Consistency for universal types in this work follows the earlier work on polymorphic gradual typing by Igarashi et al. [2017]. Their consistency relates a universal type not only to another universal type but also to what they call a non-$\forall$ type (i.e., a type such that its top type constructor is not $\forall$). The flexibility of their consistency enables interaction between statically typed code with polymorphism and dynamically typed code without polymorphism. For example, in their work, universal type $\forall X{:}\mathsf{T}.\ X \to X$ is consistent with non-$\forall$ type $\star \to \star$. Igarashi et al. present a few conditions on non-$\forall$ types to be consistent with universal types; non-$\forall$ types satisfying the conditions are called *quasi*-universal types[3] because they are not actual universal types but could behave as such by casts. We adjust their notion of quasi-universal types to our setting with row types.

Definition 3 (Quasi-universal types). *The predicate* $\mathbf{QPoly}(A)$ *is defined by:* $\mathbf{QPoly}(A)$ *if and only if (1) A is none of $\forall X{:}K.\ B$, $\cdot$ (the empty row), and $\ell : B; \rho$ for any X, K, B, ℓ, and ρ; and (2) $\star$ occurs somewhere in A. Type A is a quasi-universal type if and only if* $\mathbf{QPoly}(A)$.

Then, we introduce a consistency rule

$$\frac{\mathbf{QPoly}(A_2) \quad X \notin \mathit{ftv}(A_2) \quad A_1 \sim A_2}{\forall X{:}K.\ A_1 \sim A_2}\ \text{C\_PolyL}$$

and its symmetric version.

We make a remark on other choices of consistency for universal types. Ahmed et al. [2011, 2017] give *compatibility* instead of consistency. Their compatibility is designed to capture as many possibly successful casts as possible, and, as a result, it deems even perhaps apparently incompatible types—e.g., $\forall X{:}\mathsf{T}.\ X \to X$ and $\mathsf{int} \to \mathsf{str}$—compatible. In return for this great flexibility, their calculus lacks conservativity over typing of System F, the underlying calculus of their gradually typed language (i.e., a static typing error found by System F may not be found by their gradual type system). Another definition of consistency is given by Toro et al. [2019]. Their consistency relates a universal type only to another universal type and not to any non-$\forall$ type. Their gradually typed language achieves conservativity over typing of System F, but the strict distinction between universal types and non-$\forall$ types prevents dynamically typed code, where no type information appears, from using polymorphic values.

We follow Igarashi et al. [2017] because of its balance between flexibility—it allows dynamically typed code to use polymorphic values—and strictness—it makes a gradually typed language conservative over typing of System F. However, we believe that how to deal with universal types in consistency is orthogonal to consistency for row types and that we can choose a suitable treatment depending on cases.

[3]Igarashi et al. call such types quasi-polymorphic types, but we use that term for consistent use of terminology.

4.1.3 Formal definition. Now, we present a formal definition of consistency. We say that a relation between types is compatible if and only if it is closed under type and row constructors.

Definition 4 (Consistency). *Consistency $A \sim B$ is the smallest compatible symmetric relation satisfying (1) $\star \sim A$ for any A, (2)* (C_ConsL)*, and (3)* (C_PolyL)*.*

4.1.4 Consistency issues. Consistency does not subsume type-and-row equivalence. Thus, if a gradually typed language employed consistency directly, it would be combined with type-and-row equivalence, particularly in the form of composition $\equiv \circ \sim$. However, use of that composition gives rise to two issues, which were first found in the work on gradual typing for subtyping [Siek and Taha 2007].

The first issue is on typechecking. A typechecking algorithm for a type system using $\equiv \circ \sim$ for type comparison would have to decide whether given two types A and B are in $\equiv \circ \sim$. Thus, it would need to find an intermediate type C such that $A \equiv C$ and $C \sim B$. But, how? This issue may not be as serious as the case of subtyping [Siek and Taha 2007] because $\equiv$ just reorders fields in a row, but it should be still resolved.

The second issue is more serious: incoherent semantics. For example, let us consider the following gradually typed term:

$$M \stackrel{\text{def}}{=} \{\ell_1 = \Lambda X{:}K.\, M_1; \{\ell_2 = \Lambda X{:}K.\, M_2; \{\}\}\} : \star$$

where we suppose that ℓ_1 and ℓ_2 are distinct, M_1 is a divergent term, and M_2 is a term involving run-time checking that always fails; ascription $M' : A$ is a shorthand of $(\lambda x{:}A.x)\, M'$. In this example, the record value is injected into the dynamic type $\star$. In the course of the injection, each field value would be also injected into $\star$ so that it can be used in dynamically typed code. The problem here is that (1) under the semantics of earlier polymorphic gradually typed languages [Ahmed et al. 2011, 2017; Igarashi et al. 2017], the evaluation result of M changes depending on which field value is injected into $\star$ first and (2) the use of $\equiv \circ \sim$ prevents determining the order of the injections to be unique. Let us start with seeing the first observation. In the semantics of earlier work on polymorphic gradual typing [Ahmed et al. 2011, 2017; Igarashi et al. 2017], injection of the type abstractions $\Lambda X{:}K.\, M_1$ and $\Lambda X{:}K.\, M_2$ into $\star$ reduces to terms containing $M_1\,[\star/X]$ and $M_2\,[\star/X]$ as redexes, respectively. Thus, if $\Lambda X{:}K.\, M_1$ is injected first, the evaluation result would be divergence since M_1 is a divergent term; otherwise, if $\Lambda X{:}K.\, M_2$ is first, the result would be a failure of run-time checking since M_2 contains a failing check. Therefore, in order for the semantics to be coherent, the order of injections of $\Lambda X{:}K.\, M_1$ and $\Lambda X{:}K.\, M_2$ into $\star$ has to be unique. However, a gradual type system employing $\equiv \circ \sim$ could not determine the order to be unique. Why not? In gradual typing, how to inject values into $\star$ is decided by instances of consistency appearing in a typing derivation. In the example term M, composition $\equiv \circ \sim$ would be used to compare $[\ell_1 : \forall X{:}K.\, A; \ell_2 : \forall X{:}K.\, B; \cdot]$ and $\star$ (where A and B are types of M_1 and M_2, respectively), and there are two possible instances of consistency to derive $[\ell_1 : \forall X{:}K.\, A; \ell_2 : \forall X{:}K.\, B; \cdot]$ $(\equiv \circ \sim)$ $\star$: one is $[\ell_1 : \forall X{:}K.\, A; \ell_2 : \forall X{:}K.\, B; \cdot] \sim \star$ and the other is $[\ell_2 : \forall X{:}K.\, B; \ell_1 : \forall X{:}K.\, A; \cdot] \sim \star$. If a typing derivation with the former instance is given, $\Lambda X{:}K.\, M_1$ would be injected into $\star$ first; otherwise, if one with the latter instance is given, $\Lambda X{:}K.\, M_2$ would be first—thus, the evaluation result of M depends on which consistency instance a given typing derivation has. Although we might be able to design coherent semantics with respect to choice of consistency instances, we take another approach, consistent equivalence, which seems more standard in gradual typing [Siek and Taha 2007; Xie et al. 2018].

4.2 Consistent equivalence

To resolve the issues on consistency, we give *consistent equivalence* $\simeq$, which characterizes composition of consistency and type-and-row equivalence. Our idea is to extend the consistency rule

(C_ConsL) for row extension in such a way as to take into account when a label used for extension on the left-hand side does and does not appear in a row on the right-hand side. (The rule (C_ConsL) in Section 4.1.1 handles only the latter case.) A promising rule that handles only the former case is:

$$\frac{\rho_2 \equiv \ell : B; \rho_2' \quad A \simeq B \quad \rho_1 \sim \rho_2'}{\ell : A; \rho_1 \simeq \rho_2}$$

We merge this rule and (C_ConsL) into a single rule as follows. First, we split ρ_2 into the first field labeled with ℓ and the remaining row; these field and row correspond to $\ell : B$ and ρ_2' in the rule above, respectively. Even in the case that ρ_2 includes no field labeled with ℓ, *if* ρ_2 *ends with* $\star$, then we can *suppose* that ρ_2 includes a ℓ field because the dynamic row type can be supposed to be any row. Since we cannot know what type such a missing ℓ field has, we regard the type as $\star$ conservatively. Finally, we check consistency between A and the type of the ℓ field extracted from ρ_2 and between ρ_1 and the remaining row.

The idea above is formalized by the following consistent equivalence rule, which subsumes even the compatibility rule for row extension (shown in the beginning of Section 4.1.1):

$$\frac{\rho_2 \triangleright_\ell B, \rho_2' \quad A \simeq B \quad \rho_1 \simeq \rho_2'}{\ell : A; \rho_1 \simeq \rho_2}\ \text{CE\_ConsL}$$

where $\rho_1 \triangleright_\ell A, \rho_2$ is a formalization of the "split" operation on row types, defined as follows.

Definition 5 (Row splitting). *Row splitting* $\rho_1 \triangleright_\ell A, \rho_2$ *is defined as follows.*

$$\star \triangleright_\ell \star, \star \qquad \ell : A; \rho \triangleright_\ell A, \rho \qquad \ell' : B; \rho_1 \triangleright_\ell A, (\ell' : B; \rho_2) \quad (\text{if } \ell \neq \ell' \text{ and } \rho_1 \triangleright_\ell A, \rho_2)$$

Definition 6 (Consistent equivalence). *Consistent equivalence* $A \simeq B$ *is the smallest compatible symmetric relation satisfying (1)* $\star \simeq A$ *for any A, (2) (*CE_ConsL*), and (3) the rule of the same form as (*C_PolyL*).*

We can confirm that consistent equivalence subsumes both consistency and type-and-row equivalence by examples. For example, $\ell_1 : \mathsf{int}; \star \simeq \ell_2 : \mathsf{str}; \star$ and $(\ell_1 : A; \ell_2 : B; \cdot) \simeq (\ell_2 : B; \ell_1 : A; \cdot)$ are derivable if $\ell_1 \neq \ell_2$. More generally, it subsumes the composition of consistency and type-and-row equivalence. We can show (and indeed have shown) that $\simeq$ coincides with $\equiv \circ \sim$, but, following Xie et al. [2018], we prove another form of equivalence between $\simeq$ and combination of $\equiv$ and $\sim$; the statement in this form expects us to incorporate implicit higher-order polymorphism easily.

Theorem 4.1. *$A \simeq B$ if and only if $A \equiv A'$ and $A' \sim B'$ and $B' \equiv B$ for some A' and B'.*

We can develop a row-polymorphic gradually typed language easily by using consistent equivalence (we give it in the supplementary material). The language does not rest on consistency and, therefore, does not cause the issues on typechecking nor semantics raised by consistency. A typechecking algorithm for that language does not need to infer an intermediate type because it is enough to check if given two types are in a single relation, consistent equivalence. At first glance, one might consider that it is problematic that consistent equivalence is not syntax-directed. For example, when we would like to show $(\ell_1 : A; \ell_2 : B; \cdot) \simeq (\ell_2 : B; \ell_1 : A; \cdot)$, it may appear unclear which rule of (CE_ConsL) and its symmetric version should be applied first. Fortunately, either is fine, which is shown by the following inversion lemma together with symmetry of consistent equivalence.

Lemma 4.2. *If $\ell : A; \rho_1 \simeq \rho_2$, then $\rho_2 \triangleright_\ell B, \rho_2'$ and $A \simeq B$ and $\rho_1 \simeq \rho_2'$.*

For semantics, use of consistent equivalence makes the typing rules syntax-directed and, therefore, derivations for a typing judgment and instances of consistent equivalence appearing there are determined uniquely.

Blame labels p, q **Type-and-row names** α **Conversion labels** $\Phi ::= +\alpha \mid -\alpha$
Types and rows $A, B, C, D, \rho ::= X \mid \alpha \mid \star \mid \iota \mid A \to B \mid \forall X{:}K.\, A \mid [\rho] \mid \langle\rho\rangle \mid \cdot \mid \ell : A; \rho$
Ground types $G, H ::= \alpha \mid \iota \mid \star \to \star \mid [\star] \mid \langle\star\rangle$
Ground row types $\gamma ::= \alpha \mid \cdot \mid \ell : \star; \star$
Terms $e ::= x \mid \kappa \mid \lambda x{:}A.e \mid e_1\ e_2 \mid \Lambda X{:}K.e :: A \mid e\,A \mid$
$\{\} \mid \{\ell = e_1; e_2\} \mid \mathsf{let}\ \{\ell = x; y\} = e_1\ \mathsf{in}\ e_2 \mid$
$\ell\ e \mid \mathsf{case}\ e\ \mathsf{with}\ \langle \ell\ x \to e_1; y \to e_2\rangle \mid \uparrow\langle \ell : A\rangle\ e \mid$
$e : A \overset{p}{\Rightarrow} B \mid e : A \overset{\Phi}{\Rightarrow} B \mid \mathsf{blame}\ p$
Values $v ::= \kappa \mid \lambda x{:}A.e \mid \Lambda X{:}K.e :: A \mid \{\} \mid \{\ell = v_1; v_2\} \mid \ell\ v \mid \uparrow\langle \ell : A\rangle\ v \mid$
$v : G \overset{p}{\Rightarrow} \star \mid v : [\gamma] \overset{p}{\Rightarrow} [\star] \mid v : \langle\gamma\rangle \overset{p}{\Rightarrow} \langle\star\rangle \mid$
$v : A \overset{-\alpha}{\Rightarrow} \alpha \mid v : [\rho] \overset{-\alpha}{\Rightarrow} [\alpha] \mid v : \langle\rho\rangle \overset{-\alpha}{\Rightarrow} \langle\alpha\rangle$
Evaluation contexts $E ::= \ldots \mid E : A \overset{p}{\Rightarrow} B \mid E : A \overset{\Phi}{\Rightarrow} B$ **Name stores** $\Sigma ::= \emptyset \mid \Sigma, \alpha{:}K := A$

Fig. 4. Syntax of $\mathrm{F}^{\rho}_{\mathrm{C}}$.

5 BLAME CALCULUS $\mathrm{F}^{\rho}_{\mathrm{C}}$

This section defines a polymorphic blame calculus $\mathrm{F}^{\rho}_{\mathrm{C}}$ equipped with row types, record and variant types, and row polymorphism. As earlier polymorphic blame calculi [Ahmed et al. 2011, 2017; Igarashi et al. 2017; Toro et al. 2019], our calculus is designed so that parametricity holds. In fact, our calculus is a variant of λB by Ahmed et al. [2017], but it differs from λB in two points. First, the behavior of casts for universal types follow Igarashi et al. [2017]. Second, more importantly, $\mathrm{F}^{\rho}_{\mathrm{C}}$ deals with casts for record and variant types. In what follows, after defining the syntax, we show the type system of $\mathrm{F}^{\rho}_{\mathrm{C}}$ and then present the semantics.

5.1 Syntax

The syntax of $\mathrm{F}^{\rho}_{\mathrm{C}}$ is presented in Figure 4, where the parts overlapping with that of F^{ρ} are displayed in gray. To explain some extended parts, we first review run-time enforcement of parametricity by Ahmed et al. [2011, 2017]. After that, we detail the extended syntax of $\mathrm{F}^{\rho}_{\mathrm{C}}$.

5.1.1 Run-time enforcement of parametricity. Ahmed et al. [2011] found that type application with normal substitution-based semantics breaks parametricity. To recover parametricity in gradual typing, Ahmed et al. [2017] give a semantics that type application $(\Lambda X{:}\mathsf{T}.\ e)\ A$ generates a fresh type name α and substitutes α for X in e, where type name α works like an abstract, "fresh base type": if a value of type α is injected to the dynamic type, the resulting value can be projected successfully only to α and projection to other types always fails. While abstract inside e, α should be visible as A outside e. Ahmed et al. [2017] control such revelation and concealment of actual type information A of α by *explicit type conversion*. With a global store mapping α to A, conversion $e : B \overset{+\alpha}{\Rightarrow} C$ reveals actual type A of α in type B of term e. By contrast, conversion $e : B \overset{-\alpha}{\Rightarrow} C$ conceals A in B by α. Type C is the result of the revelation or concealment. For example, let us consider type application of $\mathsf{Id}_{\mathsf{int}} \overset{\mathrm{def}}{=} \Lambda X{:}\mathsf{T}.\ \lambda x{:}X.(x : \star) : \mathsf{int}$ which would otherwise break parametricity. In Ahmed et al.'s semantics, application $\mathsf{Id}_{\mathsf{int}}\ A\ v$ (where v is a value of A) is evaluated as follows:

$$\begin{aligned}
\mathsf{Id}_{\mathsf{int}}\ A\ v &\longrightarrow ((\lambda x{:}X.(x : \star) : \mathsf{int})[\alpha/X] : \alpha \to \mathsf{int} \overset{+\alpha}{\Rightarrow} A \to \mathsf{int})\ v \\
&\longrightarrow^{*} ((x : \star) : \mathsf{int})[v : A \overset{-\alpha}{\Rightarrow} \alpha/x] : \mathsf{int} \overset{+\alpha}{\Rightarrow} \mathsf{int} \\
&= (((v : A \overset{-\alpha}{\Rightarrow} \alpha) : \star) : \mathsf{int}) : \mathsf{int} \overset{+\alpha}{\Rightarrow} \mathsf{int}.
\end{aligned}$$

The type application generates a fresh type name α, substitutes it for bound type variable X, and reveals A to the outside (here, function application to v) by conversion $\alpha \to \mathsf{int} \overset{+\alpha}{\Rightarrow} A \to \mathsf{int}$. Applied to argument v, the conversion conceals the type A of v by α, as $v : A \overset{-\alpha}{\Rightarrow} \alpha$, and passes the abstracted value to the original function $\lambda x{:}\alpha.(x : \star) : \mathsf{int}$ (reduction from the first to the second line). From the result in the third line, we can find that it will be tested if $v : A \overset{-\alpha}{\Rightarrow} \alpha$ is an integer value. Since type name α works like a fresh base type and matches only with α itself, that test will fail whatever A is–even if $A = \mathsf{int}$. Therefore, $\mathsf{Id}_{\mathsf{int}}$ behaves uniformly—raises an exception—whatever type is substituted for X. Our blame calculus $\mathrm{F}^{\rho}_{\mathrm{C}}$ applies this idea for row parametricity as well.

5.1.2 The extended syntax of F^{ρ}_{C}. Types and rows are augmented with type-and-row names, ranged over by α. Ground types, ranged over by G and H, are type tags given to a value injected to the dynamic type. Similarly, ground row types, ranged over by γ, are row tags given to a row injected to the dynamic row type, being a row name, the empty row, or a row extension of the form $\ell : \star; \star$.

Terms, ranged over by e, have three additional constructors. A cast $e : A \overset{p}{\Rightarrow} B$ between consistently equivalent types A and B checks if the value of e can behave as B at run time. Blame label p represents the location of the cast. A conversion $e : A \overset{\Phi}{\Rightarrow} B$ with conversion label Φ conceals or reveals type information by the type name of Φ. Blame "blame p" is an (uncatchable) exception indicating failure of a cast with p. We write $e : A \overset{p}{\Rightarrow} B \overset{q}{\Rightarrow} C$ for $(e : A \overset{p}{\Rightarrow} B) : B \overset{q}{\Rightarrow} C$ and $e : A \overset{\Phi_1}{\Rightarrow} B \overset{\Phi_2}{\Rightarrow} C$ for $(e : A \overset{\Phi_1}{\Rightarrow} B) : B \overset{\Phi_2}{\Rightarrow} C$. Evaluation contexts, ranged over by E, are also extended with casts and conversions. Type abstraction $\Lambda X{:}K.e :: A$ is augmented with the type A of e.

Values, ranged over by v, have six additional constructors: the first three values are injections into $\star$, $[\star]$, and $\langle\star\rangle$ with tag G, $[\gamma]$, and $\langle\gamma\rangle$, respectively. The next three values are conversions that conceal A or ρ by α.

It is notable that embedding $\uparrow \langle \ell : A \rangle\ v$ is a value even if embedded value v is injection $\ell'\ v'$ where $\ell' \neq \ell$, while in $\mathrm{F}^{\rho}\ \ell$ and ℓ' have to be the same in order for the embedding term to be a value. This is because we would like to make the type system of $\mathrm{F}^{\rho}_{\mathrm{C}}$ syntax-directed and, for that, we drop the implicit type conversion rule (Ts_Equiv) from $\mathrm{F}^{\rho}_{\mathrm{C}}$. Thus, for example, injection $\ell\ v$ can be given type $\langle \ell : A; \ell' : B; \cdot \rangle$ but cannot be given $\langle \ell' : B; \ell : A; \cdot \rangle$ in $\mathrm{F}^{\rho}_{\mathrm{C}}$. In order to embed $\ell\ v$ into $\langle \ell' : B; \ell : A; \cdot \rangle$, we use embedding: embedding value $\uparrow \langle \ell' : B \rangle\ (\ell\ v)$ can have type $\langle \ell' : B; \ell : A; \cdot \rangle$. Conversely, if the type of value v is a variant type $\langle \ell : A; \rho \rangle$, then v must be either an injection value $\ell\ v'$ or an embedding value $\uparrow \langle \ell : A \rangle\ v'$ for some v'. Thus, the embedding operation is not only useful to make variant types easy to use in the setting with row polymorphism—this motivates Leijen [2005] to introduce the embedding operation—but also crucial to make a type system for variant types syntax-directed.

Name stores, ranged over by Σ, bind names generated during evaluation to their actual types or rows. We suppose that names bound by Σ are unique. We write $\Sigma(\alpha) = A$ if and only if $\alpha{:}K := A \in \Sigma$.

5.2 Type system

The type system of $\mathrm{F}^{\rho}_{\mathrm{C}}$ also has three judgments taking forms augmented with Σ: well-formedness judgments for typing contexts $\Sigma \vdash \Gamma$ and for types $\Sigma; \Gamma \vdash A : K$, and typing judgment $\Sigma; \Gamma \vdash e : A$. Most of the inference rules of these judgments are similar to those of F^{ρ} except for three points. First, the inference rules are also augmented with Σ. Second, new rules for the dynamic type, type-and-row names, casts, conversions, and blame are added and the typing rule for type abstractions is adapted for change of syntax; these rules are shown in Figure 5. Third, the implicit type conversion

Convertible rules $\boxed{\Sigma \vdash A \prec^{\Phi} B}$

$$\frac{name(\Phi) \neq \alpha}{\Sigma \vdash \alpha \prec^{\Phi} \alpha}\ \textsc{Cv\_TyName} \qquad \frac{\Sigma(\alpha) = A}{\Sigma \vdash \alpha \prec^{+\alpha} A}\ \textsc{Cv\_Reveal} \qquad \frac{\Sigma(\alpha) = A}{\Sigma \vdash A \prec^{-\alpha} \alpha}\ \textsc{Cv\_Conceal}$$

$$\frac{\Sigma \vdash A_2 \prec^{\overline{\Phi}} A_1 \quad \Sigma \vdash B_1 \prec^{\Phi} B_2}{\Sigma \vdash A_1 \to B_1 \prec^{\Phi} A_2 \to B_2}\ \textsc{Cv\_Fun}$$

Well-formedness rules for types and rows $\boxed{\Sigma; \Gamma \vdash A : K}$

$$\frac{\Sigma \vdash \Gamma \quad \alpha{:}K := A \in \Sigma}{\Sigma; \Gamma \vdash \alpha : K}\ \textsc{WF\_TyName} \qquad \frac{\Sigma \vdash \Gamma}{\Sigma; \Gamma \vdash \star : K}\ \textsc{WF\_Dyn}$$

Typing rules $\boxed{\Sigma; \Gamma \vdash e : A}$

$$\frac{\Sigma; \Gamma, X{:}K \vdash e : A}{\Sigma; \Gamma \vdash \Lambda X{:}K.e :: A : \forall X{:}K.\, A}\ \textsc{T\_TLam} \qquad \frac{\Sigma; \Gamma \vdash A : \mathsf{T}}{\Sigma; \Gamma \vdash \mathsf{blame}\, p : A}\ \textsc{T\_Blame}$$

$$\frac{\Sigma; \Gamma \vdash e : A \quad \Sigma; \Gamma \vdash B : \mathsf{T} \quad A \simeq B}{\Sigma; \Gamma \vdash e : A \overset{p}{\Rightarrow} B : B}\ \textsc{T\_Cast} \qquad \frac{\Sigma \vdash \Gamma \quad \Sigma; \emptyset \vdash e : A \quad \Sigma; \emptyset \vdash B : \mathsf{T} \quad \Sigma \vdash A \prec^{\Phi} B}{\Sigma; \Gamma \vdash e : A \overset{\Phi}{\Rightarrow} B : B}\ \textsc{T\_Conv}$$

Fig. 5. The type system of $\mathrm{F}^{\rho}_{\mathrm{C}}$ (selected rules).

rule (Ts_Equiv) with type-and-row equivalence is dropped and field reordering is covered by casts. Hence, the inference rules of $\mathrm{F}^{\rho}_{\mathrm{C}}$ are syntax-directed. Figure 5 shows only key rules, and the other rules have the same forms as those of F^{ρ}; interested readers can find the complete definition of the type system in the supplementary material.

There are two additional well-formedness rules for names and the dynamic type. The dynamic type $\star$ can be used as both the dynamic value type and the dynamic row type (WF_Dyn). A type-and-row name is given kind K assigned by Σ (WF_TyName).

New typing rules are added for new constructors. Types in a cast have to be consistently equivalent. A conversion $e : A \overset{\Phi}{\Rightarrow} B$ converts type A of e to type B by revealing type information $\Sigma(\alpha)$ of α in A if $\Phi = +\alpha$, or concealing it if $\Phi = -\alpha$. This idea is formalized by *convertibility* $\Sigma \vdash A \prec^{\Phi} B$, which means that, if $\Phi = +\alpha$, B is obtained by substituting $\Sigma(\alpha)$ for α in A and that, if $\Phi = -\alpha$, A is obtained by substituting $\Sigma(\alpha)$ for α in B. Convertibility is the smallest relation such that (1) it satisfies the rules given at the top of Figure 5 and (2) it is closed under type and row constructors other than names and function types. The convertibility rules use two operations on Φ: $name(\Phi)$ returns the name of Φ, i.e., $name(+\alpha) \overset{\text{def}}{=} name(-\alpha) \overset{\text{def}}{=} \alpha$; $\overline{\Phi}$ is the negation of Φ, i.e., $\overline{+\alpha} \overset{\text{def}}{=} -\alpha$ and $\overline{-\alpha} \overset{\text{def}}{=} +\alpha$. The rules (Cv_Reveal) and (Cv_Conceal) reflect the above intuition of convertibility. The rule (Cv_TyName) means that type information of $name(\Phi)$ must be revealed or concealed. The rule (Cv_Fun) means that convertibility is contravariant on argument types with the negated Φ and covariant on return types with Φ.

5.3 Semantics

The semantics of $\mathrm{F}^{\rho}_{\mathrm{C}}$ consists of two relations: the reduction relation $e_1 \rightsquigarrow e_2$, which handles basic computation irrelevant to name stores, and the evaluation relation $\Sigma_1 \mid e_1 \longrightarrow \Sigma_2 \mid e_2$, which reduces a subterm, lifts blame, or handles type application with name generation.

Reduction rules $\boxed{e_1 \rightsquigarrow e_2}$

$$\kappa_1\ \kappa_2 \rightsquigarrow \zeta(\kappa_1, \kappa_2) \quad \text{R\_Const} \qquad (\lambda x{:}A.e)\ v \rightsquigarrow e[v/x] \quad \text{R\_Beta}$$

$$\text{let } \{\ell = x; y\} = \{\ell = v_1; v_2\} \text{ in } e_2 \rightsquigarrow e[v_1/x, v_2/y] \quad \text{R\_Record}$$

$$\text{case } (\ell\ v) \text{ with } \langle \ell\ x \to e_1; y \to e_2\rangle \rightsquigarrow e_1[v/x] \quad \text{R\_CaseL}$$

$$\text{case } \uparrow\langle \ell : A\rangle\ v \text{ with } \langle \ell\ x \to e_1; y \to e_2\rangle \rightsquigarrow e_2[v/y] \quad \text{R\_CaseR}$$

$$v : A \overset{p}{\Rightarrow} A \rightsquigarrow v \quad (\text{if } A = \star, \iota, \text{ or } \alpha) \quad \text{R\_Id}$$

$$v : A \overset{p}{\Rightarrow} \star \rightsquigarrow v : A \overset{p}{\Rightarrow} G \overset{p}{\Rightarrow} \star \quad \text{R\_ToDyn}$$
$$(\text{if } A \simeq G \text{ and } A \neq G \text{ and } A \neq \star \text{ and } A \neq \forall X{:}K.\, B)$$

$$v : \star \overset{p}{\Rightarrow} A \rightsquigarrow v : \star \overset{p}{\Rightarrow} G \overset{p}{\Rightarrow} A \quad \text{R\_FromDyn}$$
$$(\text{if } A \simeq G \text{ and } A \neq G \text{ and } A \neq \star \text{ and } A \neq \forall X{:}K.\, B)$$

$$v : G \overset{p}{\Rightarrow} \star \overset{q}{\Rightarrow} G \rightsquigarrow v \quad \text{R\_Ground} \qquad v : G \overset{p}{\Rightarrow} \star \overset{q}{\Rightarrow} H \rightsquigarrow \text{blame } q \quad (\text{if } G \neq H) \quad \text{R\_Blame}$$

$$v : A_1 \to B_1 \overset{p}{\Rightarrow} A_2 \to B_2 \rightsquigarrow \lambda x{:}A_2.\, v\,(x : A_2 \overset{\bar{p}}{\Rightarrow} A_1) : B_1 \overset{p}{\Rightarrow} B_2 \quad \text{R\_Wrap}$$

$$v : \forall X{:}K.\, A_1 \overset{p}{\Rightarrow} \forall X{:}K.\, A_2 \rightsquigarrow \Lambda X{:}K.(v\,X : A_1 \overset{p}{\Rightarrow} A_2) :: A_2 \quad \text{R\_Content}$$

$$v : \forall X{:}K.\, A \overset{p}{\Rightarrow} B \rightsquigarrow (v\,\star) : A[\star/X] \overset{p}{\Rightarrow} B \quad (\text{if } \mathbf{QPoly}\,(B)) \quad \text{R\_Inst}$$

$$v : A \overset{p}{\Rightarrow} \forall X{:}K.\, B \rightsquigarrow \Lambda X{:}K.(v : A \overset{p}{\Rightarrow} B) :: B \quad (\text{if } \mathbf{QPoly}\,(A)) \quad \text{R\_Gen}$$

$$v : A \overset{-\alpha}{\Rightarrow} B \overset{+\alpha}{\Rightarrow} A \rightsquigarrow v \quad (\text{if (1) } B = \alpha; \text{ (2) } B = [\alpha] \text{ and } A = [\rho]; \text{ or (3) } B = \langle\alpha\rangle \text{ and } A = \langle\rho\rangle) \quad \text{R\_CName}$$

$$v : A \overset{\Phi}{\Rightarrow} A \rightsquigarrow v \quad (\text{if } A = \star, \alpha, \iota, [\star], [\alpha], [\cdot], \langle\star\rangle, \text{ or } \langle\alpha\rangle \text{ for } \alpha \neq \mathit{name}(\Phi)) \quad \text{R\_CId}$$

$$v : A_1 \to B_1 \overset{\Phi}{\Rightarrow} A_2 \to B_2 \rightsquigarrow \lambda x{:}A_2.\, v\,(x : A_2 \overset{\bar{\Phi}}{\Rightarrow} A_1) : B_1 \overset{\Phi}{\Rightarrow} B_2 \quad \text{R\_CFun}$$

$$v : \forall X{:}K.\, A_1 \overset{\Phi}{\Rightarrow} \forall X{:}K.\, A_2 \rightsquigarrow \Lambda X{:}K.(v\,X : A_1 \overset{\Phi}{\Rightarrow} A_2) :: A_2 \quad \text{R\_CForall}$$

$$v : [\ell : A; \rho_1] \overset{\Phi}{\Rightarrow} [\ell : B; \rho_2] \rightsquigarrow \text{let } \{\ell = x; y\} = v \text{ in } \{\ell = x : A \overset{\Phi}{\Rightarrow} B; y : [\rho_1] \overset{\Phi}{\Rightarrow} [\rho_2]\} \quad \text{R\_CRExt}$$

$$v : \langle \ell : A; \rho_1\rangle \overset{\Phi}{\Rightarrow} \langle \ell : B; \rho_2\rangle \rightsquigarrow \text{case } v \text{ with } \langle \ell\ x \to \ell\,(x : A \overset{\Phi}{\Rightarrow} B); y \to \uparrow\langle \ell : B\rangle\,(y : \langle\rho_1\rangle \overset{\Phi}{\Rightarrow} \langle\rho_2\rangle)\rangle \quad \text{R\_CVar}$$

Fig. 6. Reduction rules of $\mathsf{F}^{\rho}_{\mathsf{C}}$ except casts for record and variant types.

5.3.1 Reduction except cast for records and variants. The reduction rules except cast for record and variant types are shown in Figure 6. Most of the reduction rules for casts and conversions there come from Ahmed et al. [2017]. Cast semantics for universal types follows Igarashi et al. [2017].

The first five rules are for function application, record decomposition, and case matching. The rule (R_Record) for record decomposition assumes that the first field label of a record matches with the pattern label, while the reduction rule (Rs_Record) of F^{ρ} does not assume that and looks for the ℓ field from a record. This assumption is valid in $\mathsf{F}^{\rho}_{\mathsf{C}}$ because $\mathsf{F}^{\rho}_{\mathsf{C}}$ reorders the record fields by casts so that the static assumption of (T_Record)—the first field of a decomposed record has the same label as the pattern—is ensured even at run time. Similarly, the rules (R_CaseL) and (R_CaseR) for case matching also assume that a matched term has the same label as the pattern. For variants, instead of field reordering, applications of the embedding operation are inserted.

Casts (except for record and variant types) behave as follows. Casts where both sides are the dynamic type, a base type, or a type name behave as identity functions. If a value of A is injected to the dynamic type, it is tagged with ground type G consistently equivalent to A (R_ToDyn). Conversely, if a value of $\star$ is projected to A, it will be checked if the injected value is tagged with G consistently equivalent to A (R_FromDyn). If the check succeeds, the projection returns the injected value (R_Ground); otherwise, it raises an exception (R_Blame). Casts between function types and between universal types produce a wrapper of a given value by decomposing the types.

Cast rules for records $\boxed{e_1 \rightsquigarrow e_2}$

$$v : [\rho] \overset{p}{\Rightarrow} [\rho] \rightsquigarrow v \quad (\text{if } \rho = \cdot \text{ or } \alpha) \qquad \text{R\_RId}$$

$$v : [\rho] \overset{p}{\Rightarrow} [\star] \rightsquigarrow v : [\rho] \overset{p}{\Rightarrow} [grow(\rho)] \overset{p}{\Rightarrow} [\star] \quad (\text{if } \rho \neq grow(\rho)) \qquad \text{R\_RToDyn}$$

$$v : [\gamma] \overset{p}{\Rightarrow} [\star] \overset{q}{\Rightarrow} [\rho] \rightsquigarrow v : [\gamma] \overset{q}{\Rightarrow} [\rho] \quad (\text{if } \gamma \simeq \rho) \qquad \text{R\_RFromDyn}$$

$$v : [\gamma] \overset{p}{\Rightarrow} [\star] \overset{q}{\Rightarrow} [\rho] \rightsquigarrow \text{blame } q \quad (\text{if } \gamma \not\simeq \rho) \qquad \text{R\_RBlame}$$

$$v : [\rho_1] \overset{p}{\Rightarrow} [\ell : B; \rho_2] \rightsquigarrow \{\ell = (v_1 : A \overset{p}{\Rightarrow} B); v_2 : [\rho_1'] \overset{p}{\Rightarrow} [\rho_2]\} \quad (\text{if } v \triangleright_\ell v_1, v_2 \text{ and } \rho_1 \triangleright_\ell A, \rho_1') \qquad \text{R\_RRev}$$

$$v : [\rho_1] \overset{p}{\Rightarrow} [\ell : B; \rho_2] \rightsquigarrow v : [\rho_1] \overset{p}{\Rightarrow} [\rho_1 \mathbin{@} \ell : B] \overset{p}{\Rightarrow} [\ell : B; \rho_2] \quad (\text{if } \ell \notin dom(\rho_1) \text{ and } \rho_1 \neq \star) \qquad \text{R\_RCon}$$

Fig. 7. Reduction rules for casts between record types.

Casts from a quasi-universal type to a universal type also produces a wrapper (R_Gen). Casts from a universal type to a quasi-universal type apply a given type abstraction to $\star$ (R_Inst).

The last six rules are for conversions. Revealing the concealed type A (or ρ) of a value reduces to the value itself (R_CName). If types in a conversion take the same "atomic" form, it is just like an identity function (R_CId). If types in a conversion are not atomic, a new term is constructed by decomposing a given value and applying conversion with the type subcomponents to the result.

5.3.2 Cast reduction for records. The reduction rules for record casts are given in Figure 7.

If record types in a cast are the same and their rows are the empty row or a row name, the cast behaves as an identity function (R_RId).

If a record of type $[\rho]$ is injected into the record type $[\star]$, it is tagged with a ground row type consistently equivalent to ρ (R_RToDyn). However, such a ground row type is not always determined uniquely especially if ρ is a row extension. For example, row extension $(\ell : A; \star)$ is consistently equivalent to any ground row type of the form $(\ell' : \star; \star)$. We find a ground row type from ρ by using function *grow*, which is defined as follows: $grow(\cdot) \overset{\text{def}}{=} \cdot$; $grow(\alpha) \overset{\text{def}}{=} \alpha$; and $grow(\ell : A; \rho) \overset{\text{def}}{=} \ell : \star; \star$. If $\rho = grow(\rho)$, term $v : [\rho] \overset{p}{\Rightarrow} [\star]$ is a value, not needed to reduce.

If the type of a record to be cast is $[\star]$, then the record can be supposed to be tagged with a ground row type γ. If γ is consistently equivalent to the target type ρ of a cast, the cast reduces to another cast from γ to ρ (R_RFromDyn); otherwise, if γ is not consistently equivalent to ρ, an exception is raised (R_RBlame). Note that a cast $v : [\star] \overset{p}{\Rightarrow} [\star]$ is handled by (R_RFromDyn). One might consider why reduction of cast $v : [\star] \overset{p}{\Rightarrow} [\rho]$ is not defined as (R_FromDyn) in Figure 6, that is, the reduction does not proceed as the cast first reduces to $v : [\star] \overset{p}{\Rightarrow} [grow(\rho)] \overset{p}{\Rightarrow} [\rho]$ and then tests equality of $grow(\rho)$ and the ground row type γ attached to v. We do not give such reduction because a ground row type of ρ may not be determined to be unique and, therefore, equality test of $grow(\rho)$ and γ may fail even if the record v can behave as $grow(\rho)$. For example, if $\rho = (\ell_1 : A; \ell_2 : B; \cdot)$ and the ground row type γ attached to v is $\ell_2 : \star; \star$, then $grow(\rho) = \ell_1 : \star; \star$ is different from γ (if $\ell_1 \neq \ell_2$), but the record v may hold both of fields labeled with ℓ_1 and ℓ_2. Instead of syntactic equality, use of consistent equivalence for comparison of $grow(\rho)$ and γ might work better; indeed, the cast semantics given by (R_RFromDyn) uses this approach.

The other cast reduction rules (R_RRev) and (R_RCon) are applied to a cast $v : [\rho_1] \overset{p}{\Rightarrow} [\ell : B; \rho_2]$ which tests whether record v of type $[\rho_1]$ has an ℓ field and, if so, whether the value of the ℓ field and the other fields can behave as B and ρ_2, respectively. The rule (R_RRev) handles the case that the source row type ρ_1 holds an ℓ field. In this case, and only in this case, we can find the value v_1 of the ℓ field from record v by record splitting $v \triangleright_\ell v_1, v_2$, where v_2 is the result of removing v_1

from v. The record splitting on v is defined as Definition 1. Row splitting $\rho_1 \triangleright_\ell A, \rho_1'$ returns the type A of v_1 and the row type ρ_1' for the fields of v_2. As a result, the cast reduces to a record value composed of an ℓ field holding $v_1 : A \overset{p}{\Rightarrow} B$ and record $v_2 : [\rho_1'] \overset{p}{\Rightarrow} [\rho_2]$. If an ℓ field is not found in ρ_1 (i.e., $\ell \notin dom(\rho_1)$), the rule (R_RCon) is applied. In this case, we can find that ρ_1 ends with $\star$ since $\ell \notin dom(\rho_1)$ but ρ_1 should be consistently equivalent with $\ell : B; \rho_2$. Thus, v may hold an ℓ field in the part hidden by $\star$. The reduction result tests it by the cast from $[\rho_1]$ to $[\rho_1 \mathbin{@} \ell : B]$. The row type $\rho_1 \mathbin{@} \ell : B$ is the same as ρ_1 except that $\ell : B$ is added as the last field. Formally, $\rho \mathbin{@} \ell : A$ is defined as follows.

Definition 7 (Field postpending). *Field postpending $\rho \mathbin{@} \ell : A$ is defined as follows:*

$$(\ell' : B; \rho) \mathbin{@} \ell : A \overset{\text{def}}{=} \ell' : B; (\rho \mathbin{@} \ell : A) \qquad \star \mathbin{@} \ell : A \overset{\text{def}}{=} \ell : A; \star$$

Note that we can assume that ρ_1 ends with $\star$ and, therefore, $\rho_1 \mathbin{@} \ell : B$ is well defined if the reduced term is well typed. If record v holds an ℓ field and its value can behave as type B, then the subsequent cast from $[\rho_1 \mathbin{@} \ell : B]$ to $[\ell : B; \rho_2]$ will test if the other fields of v can behave as ρ_2.

Examples. Let us consider a few examples of reduction. In what follows, we shade subterms to be reduced and underline their reduction results.

First, cast $\{\ell_1 = 0; \{\ell_2 = \mathsf{true}; \{\}\}\} : [\ell_1 : \mathsf{int}; \ell_2 : \mathsf{bool}; \cdot] \overset{p}{\Rightarrow} [\star]$ reduces as follows:

$$\begin{aligned}
&\{\ell_1 = 0; \{\ell_2 = \mathsf{true}; \{\}\}\} : [\ell_1 : \mathsf{int}; \ell_2 : \mathsf{bool}; \cdot] \overset{p}{\Rightarrow} [\star] \\
\longrightarrow\ & \underline{\{\ell_1 = 0; \{\ell_2 = \mathsf{true}; \{\}\}\} : [\ell_1 : \mathsf{int}; \ell_2 : \mathsf{bool}; \cdot] \overset{p}{\Rightarrow} [\ell_1 : \star; \star] \overset{p}{\Rightarrow} [\star]} \\
\longrightarrow\ & \underline{\{\ell_1 = 0 : \mathsf{int} \overset{p}{\Rightarrow} \star; \{\ell_2 = \mathsf{true}; \{\}\} : [\ell_2 : \mathsf{bool}; \cdot] \overset{p}{\Rightarrow} [\star]\}} : [\ell_1 : \star; \star] \overset{p}{\Rightarrow} [\star] \\
\longrightarrow\ & \{\ell_1 = 0 : \mathsf{int} \overset{p}{\Rightarrow} \star; \underline{\{\ell_2 = \mathsf{true}; \{\}\} : [\ell_2 : \mathsf{bool}; \cdot] \overset{p}{\Rightarrow} [\ell_2 : \star; \star] \overset{p}{\Rightarrow} [\star]}\} : [\ell_1 : \star; \star] \overset{p}{\Rightarrow} [\star] \\
\longrightarrow\ & \{\ell_1 = 0 : \mathsf{int} \overset{p}{\Rightarrow} \star; \underline{\{\ell_2 = \mathsf{true} : \mathsf{bool} \overset{p}{\Rightarrow} \star; \{\} : [\cdot] \overset{p}{\Rightarrow} [\star]\}} : [\ell_2 : \star; \star] \overset{p}{\Rightarrow} [\star]\} : [\ell_1 : \star; \star] \overset{p}{\Rightarrow} [\star]
\end{aligned}$$

where a term in an odd-numbered line reduces by (R_RToDyn) and one in an even-numbered line by (R_RRev).

In order to access to an ℓ field of the above reduction result, we have to project it to, e.g., record type $[\ell : A; \star]$. The result can be written $v : [\ell_1 : \star; \star] \overset{p}{\Rightarrow} [\star]$ where

$$v' \overset{\text{def}}{=} \{\ell_2 = \mathsf{true} : \mathsf{bool} \overset{p}{\Rightarrow} \star; \{\} : [\cdot] \overset{p}{\Rightarrow} [\star]\} \qquad v \overset{\text{def}}{=} \{\ell_1 = 0 : \mathsf{int} \overset{p}{\Rightarrow} \star; v' : [\ell_2 : \star; \star] \overset{p}{\Rightarrow} [\star]\}.$$

Then, $v : [\ell_1 : \star; \star] \overset{p}{\Rightarrow} [\star] \overset{q}{\Rightarrow} [\ell : A; \star] \longrightarrow v : [\ell_1 : \star; \star] \overset{q}{\Rightarrow} [\ell : A; \star]$ by (R_RFromDyn).

If $\ell = \ell_1$, then the result reduces to:

$$\{\ell_1 = 0 : \mathsf{int} \overset{p}{\Rightarrow} \star \overset{q}{\Rightarrow} A; v' : [\ell_2 : \star; \star] \overset{p}{\Rightarrow} [\star] \overset{q}{\Rightarrow} [\star]\}$$

by (R_RRev). Thus, if $A = \mathsf{int}$, we can extract the integer value held by the ℓ_1 field in v. Otherwise, if $A \neq \mathsf{int}$, an exception blame q will be raised.

Let us return to reduction of $v : [\ell_1 : \star; \star] \overset{q}{\Rightarrow} [\ell : A; \star]$. If $\ell \neq \ell_1$, then:

$$\begin{aligned}
&v : [\ell_1 : \star; \star] \overset{q}{\Rightarrow} [\ell : A; \star] \\
\longrightarrow\ & \underline{v : [\ell_1 : \star; \star] \overset{q}{\Rightarrow} [\ell_1 : \star; \ell : A; \star] \overset{q}{\Rightarrow} [\ell : A; \star]} && \text{(R_RCon)} \\
\longrightarrow^*\ & \underline{\{\ell_1 = 0 : \mathsf{int} \overset{p}{\Rightarrow} \star; v' : [\ell_2 : \star; \star] \overset{p}{\Rightarrow} [\star] \overset{q}{\Rightarrow} [\ell : A; \star]\}} : [\ell_1 : \star; \ell : A; \star] \overset{q}{\Rightarrow} [\ell : A; \star] \\
\longrightarrow\ & \{\ell_1 = 0 : \mathsf{int} \overset{p}{\Rightarrow} \star; \underline{v' : [\ell_2 : \star; \star] \overset{q}{\Rightarrow} [\ell : A; \star]}\} : [\ell_1 : \star; \ell : A; \star] \overset{q}{\Rightarrow} [\ell : A; \star] && \text{(R_RFromDyn)}.
\end{aligned}$$

As in the case of $\ell = \ell_1$, if $\ell = \ell_2$ and $A = \mathsf{bool}$, we can extract the Boolean value held by the ℓ_2 field in v; if $\ell = \ell_2$ but $A \neq \mathsf{bool}$, an exception blame q will be raised. If $\ell \neq \ell_2$, the last shaded

Cast and conversion reduction rules for variants $\boxed{e_1 \rightsquigarrow e_2}$

$$v : \langle \alpha \rangle \overset{p}{\Rightarrow} \langle \alpha \rangle \rightsquigarrow v \qquad \text{R\_VIdName}$$

$$v : \langle \rho \rangle \overset{p}{\Rightarrow} \langle \star \rangle \rightsquigarrow v : \langle \rho \rangle \overset{p}{\Rightarrow} \langle grow(\rho) \rangle \overset{p}{\Rightarrow} \langle \star \rangle \quad (\text{if } \rho \neq grow(\rho)) \qquad \text{R\_VToDyn}$$

$$v : \langle \gamma \rangle \overset{p}{\Rightarrow} \langle \star \rangle \overset{q}{\Rightarrow} \langle \rho \rangle \rightsquigarrow v : \langle \gamma \rangle \overset{q}{\Rightarrow} \langle \rho \rangle \quad (\text{if } \gamma \simeq \rho) \qquad \text{R\_VFromDyn}$$

$$v : \langle \gamma \rangle \overset{p}{\Rightarrow} \langle \star \rangle \overset{q}{\Rightarrow} \langle \rho \rangle \rightsquigarrow \text{blame } q \quad (\text{if } \gamma \not\simeq \rho) \qquad \text{R\_VBlame}$$

$$(\ell\ v) : \langle \ell : A; \rho_1 \rangle \overset{p}{\Rightarrow} \langle \rho_2 \rangle \rightsquigarrow \uparrow \rho_{21}\, (\ell\, (v : A \overset{p}{\Rightarrow} B)) \quad (\text{if } \rho_2 = \rho_{21} \odot (\ell : B; \cdot) \odot \rho_{22} \text{ and } \ell \notin dom(\rho_{21})) \qquad \text{R\_VRevInj}$$

$$(\uparrow \langle \ell : A \rangle\ v) : \langle \ell : A; \rho_1 \rangle \overset{p}{\Rightarrow} \langle \rho_2 \rangle \rightsquigarrow \downarrow^{\rho_{21}}_{\langle \ell : B \rangle} (v : \langle \rho_1 \rangle \overset{p}{\Rightarrow} \langle \rho_{21} \odot \rho_{22} \rangle) \quad (\text{if } \rho_2 = \rho_{21} \odot (\ell : B; \cdot) \odot \rho_{22} \text{ and } \ell \notin dom(\rho_{21})) \qquad \text{R\_VRevLift}$$

$$(\ell\ v) : \langle \ell : A; \rho_1 \rangle \overset{p}{\Rightarrow} \langle \rho_2 \rangle \rightsquigarrow \uparrow \rho_2\, (\ell\ v : \langle \ell : A; \star \rangle \overset{p}{\Rightarrow} \langle \star \rangle) \quad (\text{if } \ell \notin dom(\rho_2) \text{ and } \rho_2 \neq \star) \qquad \text{R\_VConInj}$$

$$(\uparrow \langle \ell : A \rangle\ v) : \langle \ell : A; \rho_1 \rangle \overset{p}{\Rightarrow} \langle \rho_2 \rangle \rightsquigarrow (\downarrow^{\rho_2}_{\langle \ell : A \rangle} (v : \langle \rho_1 \rangle \overset{p}{\Rightarrow} \langle \rho_2 \rangle)) : \langle \rho_2 \mathbin{@} \ell : A \rangle \overset{p}{\Rightarrow} \langle \rho_2 \rangle \quad (\text{if } \ell \notin dom(\rho_2) \text{ and } \rho_2 \neq \star) \qquad \text{R\_VConLift}$$

Fig. 8. Reduction rules for casts between variant types.

part in turn evaluates to:

$$\begin{array}{lll} & v' : [\ell_2 : \star; \star] \overset{q}{\Rightarrow} [\ell_2 : \star; \ell : A; \star] \overset{q}{\Rightarrow} [\ell : A; \star] & \text{(R\_RCon)} \\ \longrightarrow^* & \{\ell_2 = \text{true} : \text{bool} \overset{p}{\Rightarrow} \star; \{\} : [\cdot] \overset{p}{\Rightarrow} [\star] \overset{q}{\Rightarrow} [\ell : A; \star]\} : [\ell_2 : \star; \ell : A; \star] \overset{q}{\Rightarrow} [\ell : A; \star] & \\ \longrightarrow^* & \text{blame } q & \text{(R\_RBlame)} \end{array}$$

This behavior is expected because v does not hold any field with label ℓ other than ℓ_1 and ℓ_2.

5.3.3 Cast reduction for variants. The reduction rules for casts between variant types are given in Figure 8. The first four rules are similar to ones for records. The other four rules are for a cast from $\langle \ell : A; \rho_1 \rangle$ to $\langle \rho_2 \rangle$ where $\rho_2 \neq \star$. We can suppose that the cast variant value is an injection tagged with ℓ or an embedding value with $\ell : A$ under the assumption that it is typed at $\langle \ell : A; \rho_1 \rangle$.

The rules (R_VRevInj) and (R_VRevLift) are applied if ρ_2 holds an ℓ field. We use *row concatenation* to split ρ_2 into the preceding fields ρ_{21} such that $\ell \notin dom(\rho_{21})$, the first ℓ field with type B, and the following fields ρ_{22} after the ℓ field. Row concatenation $\odot$ is defined by: $(\ell_1 : A_1; \ldots; \ell_n : A_n; \cdot) \odot \rho_2 = \ell_1 : A_1; \ldots; \ell_n : A_n; \rho_2$.

If the cast variant value is an injection $\ell\ v$, the cast reduces by (R_VRevInj). Since the target variant type $\langle \rho_2 \rangle$ requires a value injected with ℓ to be typed at B, the reduction result injects the result of casting v to B with ℓ. Furthermore, the injection $\ell\, (v : A \overset{p}{\Rightarrow} B)$, which can be typed at $\langle \ell : B; \rho_{22} \rangle$, is embedded into $\langle \rho_{21} \odot (\ell : B; \cdot) \odot \rho_{22} \rangle = \langle \rho_2 \rangle$ by a sequence of applications of the embedding operation with fields in ρ_{21}, which is defined as follows.

Definition 8 (Row embedding). *Row embedding $\uparrow \rho\ e$ is defined as follows:*

$$\uparrow (\ell : A; \rho)\ e \overset{\text{def}}{=} \uparrow \langle \ell : A \rangle\, (\uparrow \rho\ e) \qquad \uparrow \rho\ e \overset{\text{def}}{=} e \quad \textit{(if } \rho \textit{ is not a row extension)}$$

The rule (R_VRevLift) is applied if the cast variant value is an embedding value $\uparrow \langle \ell : A \rangle\ v$. In this case, the field $\ell : B$ in ρ_2 is inserted by applying the embedding operation to the result of casting v to the variant type $\langle \rho_{21} \odot \rho_{22} \rangle$ with the other fields. The insertion of field $\ell : B$ is performed by the following operation.

Definition 9 (Field insertion). *Function* $\downarrow^{\rho}_{\langle \ell : A \rangle} e$ *embeds a term* e *of type* $\langle \rho \odot \rho' \rangle$ *into* $\langle \rho \odot (\ell : A; \cdot) \odot \rho' \rangle$. *Formally, it is defined as follows:*

$$\downarrow^{(\ell' : B'; \rho)}_{\langle \ell : A \rangle} e \stackrel{\text{def}}{=} \mathsf{case}\ e\ \mathsf{with}\ \langle \ell'\ x \to \ell'\ x; y \to \uparrow \langle \ell' : B' \rangle\ (\downarrow^{\rho}_{\langle \ell : A \rangle} y) \rangle$$
$$\downarrow^{\rho}_{\langle \ell : A \rangle} e \stackrel{\text{def}}{=} \uparrow \langle \ell : A \rangle\ e \quad \textit{(if } \rho \textit{ is not a row extension)}$$

Row embedding $\uparrow \rho\ e$ is justified as follows. If ρ is not a row extension (i.e., it is the empty row), then e is typed at $\langle \rho \odot \rho' \rangle = \langle \rho' \rangle$ and, therefore, $\uparrow \langle \ell : A \rangle\ e$ has type $\langle \ell : A; \rho' \rangle = \langle \rho \odot (\ell : A; \cdot) \odot \rho' \rangle$. If ρ is row extension $\ell' : B'; \rho''$, then it is checked whether e is an injection or an embedding term with ℓ' by case matching. If e is an injection, it can have type $\langle (\ell' : B'; \rho'') \odot (\ell : A; \cdot) \odot \rho' \rangle$ because in general an injection with ℓ can be typed at $\langle \ell : A; \rho \rangle$ for any row ρ. Otherwise, if e is an embedding term, the embedded value y is typed at $\langle \rho'' \odot \rho' \rangle$. Thus, row embedding is recursively applied to embed y into $\langle \rho'' \odot (\ell : A; \cdot) \odot \rho' \rangle$, and then, the embedding operation with $\ell' : B'$ is applied to the result in order to embed it into $\langle (\ell' : B'; \rho'') \odot (\ell : A; \cdot) \odot \rho' \rangle$.

The last two rules (R_VConInj) and (R_VConLift) are for the case that ρ_2 does not hold an ℓ field. In this case, we can suppose that ρ_2 ends with $\star$ under the assumption that the reduced term is well typed. If the cast variant value is an injection $\ell\ v$, it is cast to the variant type $\langle \star \rangle$ and then embedded into type $\langle \rho_2 \rangle$ (R_VConInj). If the cast value is an embedding value $\uparrow \langle \ell : A \rangle\ v$, the embedded value v is cast to $\langle \rho_2 \rangle$ and field $\ell : A$ is inserted, and then the result is cast to $\langle \rho_2 \rangle$ again (R_VConLift). The field insertion is necessary for dynamic gradual guarantee [Siek et al. 2015]; in this paper we do not prove that property, but we will show the need of the field insertion by examples in the following.

Examples. First, let us consider cast $(\uparrow \langle \ell_2 : \mathsf{bool} \rangle\ (\ell_1\ 0)) : \langle \ell_2 : \mathsf{bool}; \ell_1 : \mathsf{int}; \cdot \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle$, which reduces as follows.

$$\begin{array}{lll}
 & (\uparrow \langle \ell_2 : \mathsf{bool} \rangle\ (\ell_1\ 0)) : \langle \ell_2 : \mathsf{bool}; \ell_1 : \mathsf{int}; \cdot \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle & \\
\longrightarrow & \underline{\uparrow \langle \ell_2 : \mathsf{bool} \rangle\ (\ell_1\ 0) : \langle \ell_2 : \mathsf{bool}; \ell_1 : \mathsf{int}; \cdot \rangle \stackrel{p}{\Rightarrow} \langle \ell_2 : \star; \star \rangle} \stackrel{p}{\Rightarrow} \langle \star \rangle & \text{(R_VToDyn)} \\
\longrightarrow & \underline{\uparrow \langle \ell_2 : \star \rangle\ ((\ell_1\ 0) : \langle \ell_1 : \mathsf{int}; \cdot \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle)} : \langle \ell_2 : \star; \star \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle & \text{(R_VRevLift)} \\
\longrightarrow & \uparrow \langle \ell_2 : \star \rangle\ \underline{((\ell_1\ 0) : \langle \ell_1 : \mathsf{int}; \cdot \rangle \stackrel{p}{\Rightarrow} \langle \ell_1 : \star; \star \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle)} : \langle \ell_2 : \star; \star \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle & \text{(R_VToDyn)} \\
\longrightarrow & \uparrow \langle \ell_2 : \star \rangle\ (\underline{(\ell_1\ (0 : \mathsf{int} \stackrel{p}{\Rightarrow} \star))} : \langle \ell_1 : \star; \star \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle) : \langle \ell_2 : \star; \star \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle & \text{(R_VRevInj)}
\end{array}$$

Next, let us cast the above result to $\langle \ell : A; \star \rangle$; let $v \stackrel{\text{def}}{=} (\ell_1\ (0 : \mathsf{int} \stackrel{p}{\Rightarrow} \star)) : \langle \ell_1 : \star; \star \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle$. Then,

$$(\uparrow \langle \ell_2 : \star \rangle\ v) : \langle \ell_2 : \star; \star \rangle \stackrel{p}{\Rightarrow} \langle \star \rangle \stackrel{q}{\Rightarrow} \langle \ell : A; \star \rangle \longrightarrow (\uparrow \langle \ell_2 : \star \rangle\ v) : \langle \ell_2 : \star; \star \rangle \stackrel{q}{\Rightarrow} \langle \ell : A; \star \rangle \quad (1)$$

by (R_VFromDyn).

If $\ell = \ell_2$, then (1) reduces to $\uparrow \langle \ell_2 : A \rangle\ (v : \langle \star \rangle \stackrel{q}{\Rightarrow} \langle \star \rangle)$ by (R_VRevLift). Thus, the cast just changes the type given to the embedding operation.

If $\ell \neq \ell_2$, then, by (R_VConLift), (1) reduces to:

$$(\downarrow^{\ell : A; \star}_{\langle \ell_2 : \star \rangle} (\ v : \langle \star \rangle \stackrel{q}{\Rightarrow} \langle \ell : A; \star \rangle\ )) : \langle \ell : A; \ell_2 : \star; \star \rangle \stackrel{p}{\Rightarrow} \langle \ell : A; \star \rangle. \quad (2)$$

If $\ell = \ell_1$, then the shaded part in (2) reduces to $\ell_1\ (0 : \mathsf{int} \stackrel{p}{\Rightarrow} \star \stackrel{q}{\Rightarrow} A)$ by (R_VFromDyn) and (R_VRevInj). Thus, if $A \neq \mathsf{int}$, blame q is raised; otherwise, (2) reduces to

$$(\downarrow^{\ell : A; \star}_{\langle \ell_2 : \star \rangle} \underline{(\ell_1\ 0)}) : \langle \ell : A; \ell_2 : \star; \star \rangle \stackrel{p}{\Rightarrow} \langle \ell : A; \star \rangle$$

If $\ell \neq \ell_1$, then the shaded part in (2) reduces to $\uparrow\langle \ell : A\rangle\,(\ell_1\,(0 : \mathsf{int} \overset{p}{\Rightarrow} \star) : \langle \ell_1 : \star; \star\rangle \overset{q}{\Rightarrow} \langle\star\rangle)$ by (R_VFromDyn) and (R_VConInj). Note that if $\ell : A; \star$ in the shaded part were $\ell : A; \cdot$, exception blame q would be raised by using (R_VBlame).

Finally, we show that the field insertion in (R_VConLift) is crucial to prove dynamic gradual guarantee. To confirm that, let us suppose that (R_VConLift) does not perform field insertion and instead takes the following form.

$$(\uparrow\langle \ell : A\rangle\, v) : \langle \ell : A; \rho_1\rangle \overset{p}{\Rightarrow} \langle \rho_2\rangle \;\rightsquigarrow\; v : \langle \rho_1\rangle \overset{p}{\Rightarrow} \langle \rho_2\rangle \quad \text{(R\_VConLift')}$$

As an example, consider reduction of term e given as follows:

$$\begin{array}{lll} v & \overset{\text{def}}{=} & \ell\,(0 : \mathsf{int} \overset{p_1}{\Rightarrow} \star) : \langle \ell : \star; \star\rangle \overset{p_2}{\Rightarrow} \langle\star\rangle \\ e & \overset{\text{def}}{=} & (\uparrow\langle \ell : \mathsf{bool}\rangle\, v) : \langle \ell : \mathsf{bool}; \star\rangle \overset{p_3}{\Rightarrow} \langle \ell' : \mathsf{str}; X\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool}; \star\rangle \end{array}$$

Dynamic gradual guarantee [Siek et al. 2015] states that changing types in a program to $\star$ does not change its behavior. In the case of e, it means that, if $e[(\ell : \mathsf{bool}; \star)/X]$ does not raise blame, $e[\star/X]$ does not either. First, let us reduce $e[(\ell : \mathsf{bool}; \star)/X]$.

$$\begin{array}{lll} & \underline{(\downarrow^{\ell':\mathsf{str};\cdot}_{\langle \ell:\mathsf{bool}\rangle}(v : \langle\star\rangle \overset{p_3}{\Rightarrow} \langle \ell' : \mathsf{str}; \star\rangle)) : \langle \ell' : \mathsf{str}; \ell : \mathsf{bool}; \star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool}; \star\rangle} & \text{(R\_VRevLift)} \\ \longrightarrow^* & (\downarrow^{\ell':\mathsf{str};\cdot}_{\langle \ell:\mathsf{bool}\rangle}(\uparrow\langle \ell' : \mathsf{str}\rangle\, v)) : \langle \ell' : \mathsf{str}; \ell : \mathsf{bool}; \star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool}; \star\rangle & \\ \longrightarrow^* & (\uparrow\langle \ell' : \mathsf{str}\rangle\,(\uparrow\langle \ell : \mathsf{bool}\rangle\, v)) : \langle \ell' : \mathsf{str}; \ell : \mathsf{bool}; \star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool}; \star\rangle & \text{(R\_CaseR)} \\ \longrightarrow^* & \underline{(\uparrow\langle \ell : \mathsf{bool}\rangle\, v) : \langle \ell : \mathsf{bool}; \star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool}; \star\rangle} & \text{(R\_VConLift')} \\ \longrightarrow^* & \uparrow\langle \ell : \mathsf{bool}\rangle\, v & \end{array}$$

Thus, $e[(\ell : \mathsf{bool}; \star)/X]$ evaluates to a value under use of (R_VConLift'). If dynamic gradual guarantee holds, so should $e[\star/X]$. However, it does not:

$$\begin{array}{lll} e[\star/X] \longrightarrow & \underline{v : \langle\star\rangle \overset{p_3}{\Rightarrow} \langle \ell' : \mathsf{str}; \star\rangle} \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool}; \star\rangle & \text{(R\_VConLift')} \\ \longrightarrow^* & \underline{\uparrow\langle \ell' : \mathsf{str}\rangle\, v : \langle \ell' : \mathsf{str}; \star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool}; \star\rangle} & \text{(R\_VFromDyn) and (R\_VConInj)} \\ \longrightarrow & \underline{v : \langle\star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool}; \star\rangle} & \text{(R\_VConLift')} \\ \longrightarrow^* & \underline{\ell\,(0 : \mathsf{int} \overset{p_1}{\Rightarrow} \star \overset{p_4}{\Rightarrow} \mathsf{bool})} & \text{(R\_VFromDyn) and (R\_VRevInj)} \\ \longrightarrow^* & \underline{\mathsf{blame}\ p_4} & \text{(R\_Blame)} \end{array}$$

We can confirm that both $e[(\ell : \mathsf{bool}; \star)/X]$ and $e[\star/X]$ evaluate to values if we use (R_VConLift). We show only the reduction of $e[\star/X]$; the reduction of $e[(\ell : \mathsf{bool}; \star)/X]$ is similar to the case of

Evaluation rules $\boxed{\Sigma_1 \mid e_1 \longrightarrow \Sigma_2 \mid e_2}$

$\Sigma \mid E[e_1] \longrightarrow \Sigma \mid E[e_2]$ (if $e_1 \rightsquigarrow e_2$) E_Red $\quad \Sigma \mid E[\mathsf{blame}\, p] \longrightarrow \Sigma \mid \mathsf{blame}\, p$ (if $E \neq [\,]$) E_Blame

$\Sigma \mid E[(\Lambda X{:}K.e :: A)\, B] \longrightarrow \Sigma, \alpha{:}K := B \mid E[e[\alpha/X] : A[\alpha/X] \overset{+\alpha}{\Rightarrow} A[B/X]]$ E_TyBeta

Fig. 9. Evaluation rules of $\mathrm{F}^{\rho}_{\mathrm{C}}$.

using (R_VConLift').

$$
\begin{array}{lll}
 & (\downarrow^{\ell':\mathsf{str};\star}_{\langle \ell:\mathsf{bool}\rangle}(v : \langle\star\rangle \overset{p_3}{\Rightarrow} \langle \ell' : \mathsf{str};\star\rangle)) : \langle \ell' : \mathsf{str}; \ell : \mathsf{bool};\star\rangle \overset{p_3}{\Rightarrow} \langle \ell' : \mathsf{str};\star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool};\star\rangle & \text{(R\_VConLift)} \\
\longrightarrow^* & (\downarrow^{\ell':\mathsf{str};\star}_{\langle \ell:\mathsf{bool}\rangle}(\uparrow\langle \ell' : \mathsf{str}\rangle\, v)) : \langle \ell' : \mathsf{str}; \ell : \mathsf{bool};\star\rangle \overset{p_3}{\Rightarrow} \langle \ell' : \mathsf{str};\star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool};\star\rangle & \\
\longrightarrow & (\uparrow\langle \ell' : \mathsf{str}\rangle\, (\uparrow\langle \ell : \mathsf{bool}\rangle\, v)) : \langle \ell' : \mathsf{str}; \ell : \mathsf{bool};\star\rangle \overset{p_3}{\Rightarrow} \langle \ell' : \mathsf{str};\star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool};\star\rangle & \text{(R\_CaseR)} \\
\longrightarrow & (\uparrow\langle \ell' : \mathsf{str}\rangle\, ((\uparrow\langle \ell : \mathsf{bool}\rangle\, v) : \langle \ell : \mathsf{bool};\star\rangle \overset{p_3}{\Rightarrow} \langle\star\rangle)) : \langle \ell' : \mathsf{str};\star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool};\star\rangle & \text{(R\_VRevLift)} \\
\longrightarrow & (\uparrow\langle \ell' : \mathsf{str}\rangle\, v') : \langle \ell' : \mathsf{str};\star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool};\star\rangle & \text{(R\_VToDyn) and (R\_VRevLift)} \\
 & \qquad (\text{where } v' = (\uparrow\langle \ell : \star\rangle\, v) : \langle \ell : \star;\star\rangle \overset{p_3}{\Rightarrow} \langle\star\rangle) & \\
\longrightarrow & (\downarrow^{\ell:\mathsf{bool};\star}_{\langle \ell':\mathsf{str}\rangle}(v' : \langle\star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool};\star\rangle)) : \langle \ell : \mathsf{bool}; \ell' : \mathsf{str};\star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool};\star\rangle & \text{(R\_VConLift)} \\
\longrightarrow^* & (\uparrow\langle \ell : \mathsf{bool}\rangle\, (\uparrow\langle \ell' : \mathsf{str}\rangle\, v)) : \langle \ell : \mathsf{bool}; \ell' : \mathsf{str};\star\rangle \overset{p_4}{\Rightarrow} \langle \ell : \mathsf{bool};\star\rangle & \\
\longrightarrow^* & (\uparrow\langle \ell : \mathsf{bool}\rangle\, ((\uparrow\langle \ell' : \star\rangle\, v) : \langle \ell' : \star;\star\rangle \overset{p}{\Rightarrow} \langle\star\rangle)) &
\end{array}
$$

Thus, we believe that the field insertion is key to show dynamic gradual guarantee, though it is left as future work.

5.3.4 Evaluation. The evaluation rules are shown in Figure 9. A term evaluates if its subterm under an evaluation context reduces (E_Red), triggers an exception (E_Blame), or involves type application (E_TyBeta). As discussed in Section 5.1, type application $(\Lambda X{:}K.e :: A)\, B$ generates a fresh name α, substitutes α for X in e, stores the actual type (or row) B of α in name store Σ, and reveals B to evaluation context E.

5.4 Properties

We show type soundness of $\mathrm{F}^{\rho}_{\mathrm{C}}$ via progress and subject reduction.

Theorem 5.1 (Type soundness). *If $\emptyset; \emptyset \vdash e : A$ and $\emptyset \mid e \longrightarrow^* \Sigma' \mid e'$ and e' cannot be evaluated under Σ', then either e' is a value or $e' = \mathsf{blame}\, p$ for some p.*

We also show that our surface language $\mathrm{F}^{\rho}_{\mathrm{G}}$ is conservative over typing of $\mathrm{F}^{\rho}_{\mathrm{C}}$. We omit the full presentation of $\mathrm{F}^{\rho}_{\mathrm{G}}$, but, as usual [Siek and Taha 2006], it is obtained by changing F^{ρ} so that (1) types are extended with $\star$ and (2) the typing rules use consistent equivalence instead of type equality. We write $\Gamma \vdash M : A$ if M has type A under Γ in $\mathrm{F}^{\rho}_{\mathrm{G}}$. For example, the typing rule for record decomposition in $\mathrm{F}^{\rho}_{\mathrm{G}}$ is

$$\frac{\Gamma \vdash M_1 : A \quad A \triangleright [\rho] \quad \rho \triangleright_{\ell} B, \rho' \quad \Gamma, x{:}B, y{:}[\rho'] \vdash M_2 : C}{\Gamma \vdash \mathsf{let}\, \{\ell = x; y\} = M_1\, \mathsf{in}\, M_2 : C}\ \text{Tg\_RLet}$$

where type matching $A \triangleright [\rho]$ is defined as: $\star \triangleright [\star]$ and $[\rho] \triangleright [\rho]$. Type-preserving translation $\Gamma \vdash M : A \hookrightarrow e$ from M of A under Γ in $\mathrm{F}^{\rho}_{\mathrm{G}}$ to e in $\mathrm{F}^{\rho}_{\mathrm{C}}$ is given by inserting casts where type matching and consistent equivalence are used. The full definitions are found in the supplementary material.

Theorem 5.2. *If $\Gamma \vdash M : A$, then there exists some e such that $\Gamma \vdash M : A \hookrightarrow e$ and $\emptyset; \Gamma \vdash e : A$.*

We state that the language $\mathrm{F}^{\rho}_{\mathrm{G}}$ is a conservative extension of F^{ρ} in terms of typing.

Theorem 5.3 (Conservativity over typing). *Suppose that $\star$ does not appear in Γ, A, and M. (1) If $\Gamma \vdash M : A$, then $\Gamma \vdash^{s} M : A$. (2) If $\Gamma \vdash^{s} M : A$, then $\Gamma \vdash M : B$ for some B such that $A \equiv B$.*

6 RELATED WORK

6.1 Row types, row polymorphism, and their applications

Row types were introduced by Wand [1987], who has studied type inference for objected-oriented languages and modeled objects in a variant of λ-calculus equipped with record types and variant types with rows. Wand also introduced row type variables for row type inference and discussed row polymorphism informally. Although that work supposed labels in a row type to be unique, it allowed record extension $\{\ell = M_1; M_2\}$ even for record M_2 holding an ℓ field; if M_2 contains an ℓ field, its value will be overwritten by M_1. However, this overwriting semantics causes an issue that some programs do not have principal types [Wand 1991]. Gaster and Jones [1996] resolved this issue by allowing record extension only when record M_2 does not contain an ℓ field. With help of presence and absence types [Rémy 1989], they gave a type inference algorithm that produces a principal type (if any). In order for row type substitution to preserve uniqueness of labels, they employed *qualified types* called "lacks" predicates, which constrain quantified row type variables to be instantiated only with row types that lack some fields. The use of presence and absence types also enabled them to deal with record restriction, which was not handled by Wand [1987, 1991].

Another approach to principal typing for rows is to lift the uniqueness restriction and to allow scoped labels. Scoped labels were first discussed by Berthomieu and le Moniès de Sagazan [1995] in the context of process calculi and later applied to functional programming by Leijen [2005], who also developed a sound and complete unification algorithm for inference of row types with scoped labels. In this work we adopt scoped labels, which enable us not only to simplify the metatheory of our calculus but also to use the embedding operation [Leijen 2005]. The embedding operation is helpful to align variant types with different row types in a polymorphic setting. In our work, it is also important to make the type system of $\mathrm{F}^{\rho}_{\mathrm{C}}$ syntax-directed.

Row types have been applied, e.g., to model objects [Rémy and Vouillon 1998; Wand 1987, 1991] and polymorphic variants [Garrigue 1998] and have been found in many programming languages. A more recent application of row types is an effect system for effect handlers [Plotkin and Pretnar 2009] with [Leijen 2014, 2017; Lindley et al. 2017] or without [Hillerström and Lindley 2016] scoped labels. Actually, our formalization of scoped labels and the embedding operation is influenced by Hillerström et al. [2017] and Biernacki et al. [2018], respectively.

6.2 Gradual typing for records and variants

Takikawa et al. [2012] studied gradual typing for first-class classes. They employed row types and row polymorphism for expressing presence and absence of interesting methods. Thus, they did not handle variant types and considered row polymorphism together with lacks predicates. Their work dealt with specifications written in the form of contracts and supposed contracts to play a role of interface for module components. This style of gradual typing is called "macro"-level gradual typing. i.e., typed and untyped modules are mixed, while we focus on "micro"-level gradual typing, where typed and untyped expressions are mixed. Technically, this difference appears, e.g., in the need of consistency. In addition, as our work, they also protected polymorphically typed values from untyped code. Their development, *sealing contracts*, has finer-grained control than row names in our work in that sealing contracts can expose absence of fields, while row names cannot.

Garcia et al. [2016] proposed a general framework to derive a gradually typed language from a statically typed one. They also developed *gradual rows*, which are rows possibly ending with the

dynamic row type, for record types via application of their framework to a calculus with width and depth record subtyping. Thus, a clear difference between their and our work is support for variant types and row polymorphism. The consistency relation in their work involves row equivalence, and, therefore, it seems to be equivalent to consistent equivalence given by the present work (modulo support for variant types and row polymorphism).

Jafery and Dunfield [2017] introduced dynamic sums for gradual datasort refinement. A dynamic sum $A +^{?} B$ can be interpreted as both of a single type A and B, and its value can be deconstructed by a case expression having a single branch for A or B. In our calculus dynamic sums can be encoded by two-fold variant type $\langle \ell_1 : A; \ell_2 : B; \cdot \rangle$ which are coerced to $\langle \ell_1 : A; \cdot \rangle$ or $\langle \ell_2 : B; \cdot \rangle$ in case matching via injection to $\langle \star \rangle$. Unlike our work, they did not deal with labeled fields and row polymorphism.

7 CONCLUSION

We have introduced the dynamic row type and consistency for gradual typing with row types and row polymorphism. While consistency captures the static aspect of the dynamic row type, we have found that it is problematic if combined with row equivalence. To solve the problem with consistency, we have developed consistent equivalence and shown that it characterizes composition of consistency and row equivalence. We also have given a polymorphic blame calculus $\mathrm{F}^{\rho}_{\mathrm{C}}$ with scoped labels, row types, row polymorphism, and consistent equivalence and proven its type soundness as well as type-preservation of translation from surface language $\mathrm{F}^{\rho}_{\mathrm{G}}$ to $\mathrm{F}^{\rho}_{\mathrm{C}}$ and conservativity of $\mathrm{F}^{\rho}_{\mathrm{G}}$ over typing of F^{ρ}. The cast semantics of $\mathrm{F}^{\rho}_{\mathrm{C}}$ is designed carefully to take into account criteria of gradual typing [Siek et al. 2015], but proving them is left as future work. Another direction of future work is to extend our calculus to effect systems for effect handlers [Plotkin and Pretnar 2009]. It is also interesting to "gradualize" other formalisms, such as presence and absence types [Pottier and Rémy 2005] and a general framework for row types [Morris and McKinna 2019].

ACKNOWLEDGMENTS

We would like to thank John Toman for proofreading. This work was supported in part by: JSPS KAKENHI Grant Number JP17H01723 (Igarashi); and JSPS KAKENHI Grant Number JP19K20247 and ERATO HASUO Metamathematics for Systems Design Project (JPMJER1603), JST (Sekiyama).

REFERENCES

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for all. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* 201–214. https://doi.org/10.1145/1926385.1926409

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for free for free: parametricity, with and without types. *PACMPL* 1, ICFP (2017), 39:1–39:28. https://doi.org/10.1145/3110283

Bernard Berthomieu and Camille le Moniès de Sagazan. 1995. A Calculus of Tagged Types, with applications to process languages. In *Workshop on Types for Program Analysis.* 1–15.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2018. Handle with care: relational interpretation of algebraic effects and handlers. *PACMPL* 2, POPL (2018), 8:1–8:30. https://doi.org/10.1145/3158096

Luca Cardelli and John C. Mitchell. 1991. Operations on Records. *Mathematical Structures in Computer Science* 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

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 303–315. https://doi.org/10.1145/2676726.2676992

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 429–442. https://doi.org/10.1145/2837614.2837670

Jacques Garrigue. 1998. Programming with polymorphic variants. In *In ACM Workshop on ML.*

Jacques Garrigue. 2000. Code reuse through polymorphic variants. In *In Workshop on Foundations of Software Engineering*.

Benedict R. Gaster and Mark P. Jones. 1996. A Polymorphic Type System for Extensible Records and Variants.

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*. 15–27. https://doi.org/10.1145/2976022.2976033

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*. 18:1–18:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.18

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On polymorphic gradual typing. *PACMPL* 1, ICFP (2017), 40:1–40:29. https://doi.org/10.1145/3110284

Khurram A. Jafery and Joshua Dunfield. 2017. Sums of uncertainty: refinements go gradual. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 804–817.

Daan Leijen. 2005. Extensible records with scoped labels. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005*. 179–194.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*. 100–126. https://doi.org/10.4204/EPTCS.153.8

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*. 486–499. http://dl.acm.org/citation.cfm?id=3009872

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*. 500–514. http://dl.acm.org/citation.cfm?id=3009897

Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic type inference for gradual Hindley-Milner typing. *PACMPL* 3, POPL (2019), 18:1–18:29. https://doi.org/10.1145/3290331

J. Garrett Morris and James McKinna. 2019. Abstracting extensible data types: or, rows by any other name. *PACMPL* 3, POPL (2019), 12:1–12:28. https://doi.org/10.1145/3290325

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, Proceedings*. 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

François Pottier and Didier Rémy. 2005. *Advanced Topics in Types and Programming Languages*. The MIT Press, Chapter The Essence of ML Type Inference, 387–489.

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*. 77–88. https://doi.org/10.1145/75277.75284

Didier Rémy and Jerome Vouillon. 1998. Objective ML: An Effective Object-Oriented Extension to ML. *TAPOS* 4, 1 (1998), 27–50.

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*. 408–423. https://doi.org/10.1007/3-540-06859-7_148

Taro Sekiyama, Soichiro Ueda, and Atsushi Igarashi. 2015. Shifting the Blame - A Blame Calculus with Delimited Control. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*. 189–207. https://doi.org/10.1007/978-3-319-26529-2_11

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Proc. of Workshop on Scheme and Functional Programming*. 81–92.

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual typing with unification-based inference. In *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*. 7. https://doi.org/10.1145/1408681.1408688

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*. 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual typing for first-class classes. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. 793–810. https://doi.org/10.1145/2384616.2384674

Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. 2013. Constraining Delimited Control with Contracts. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 229–248. https://doi.org/10.1007/978-3-642-37036-6_14

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. 964–974. https://doi.org/10.1145/1176617.1176755

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual parametricity, revisited. *PACMPL* 3, POPL (2019), 17:1–17:30. https://doi.org/10.1145/3290330

Mitchell Wand. 1987. Complete Type Inference for Simple Objects. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. 37–44.

Mitchell Wand. 1991. Type Inference for Record Concatenation and Multiple Inheritance. *Inf. Comput.* 93, 1 (1991), 1–15. https://doi.org/10.1016/0890-5401(91)90050-C

Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. 2018. Consistent Subtyping for All. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 3–30. https://doi.org/10.1007/978-3-319-89884-1_1