Row and Bounded Polymorphism via Disjoint Polymorphism

Ningning Xie
The University of Hong Kong, Hong Kong, China
nnxie@cs.hku.hk

Bruno C. d. S. Oliveira
The University of Hong Kong, Hong Kong, China
bruno@cs.hku.hk

Xuan Bi
The University of Hong Kong, Hong Kong, China
xbi@cs.hku.hk

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract

Polymorphism and subtyping are important features in mainstream OO languages. The most common way to integrate the two is via $\mathsf{F}_{<:}$ style *bounded quantification*. A closely related mechanism is row polymorphism, which provides an alternative to subtyping, while still enabling many of the same applications. Yet another approach is to have type systems with *intersection types* and polymorphism. A recent addition to this design space are calculi with *disjoint intersection types* and *disjoint polymorphism*. With all these alternatives it is natural to wonder how they are related.

This paper provides *an* answer to this question. We show that disjoint polymorphism can recover forms of both row polymorphism and bounded polymorphism, while retaining key desirable properties, such as type-safety and decidability. Furthermore, we identify the extra power of disjoint polymorphism which enables additional features that cannot be easily encoded in calculi with row polymorphism or bounded quantification alone. Ultimately we expect that our work is useful to inform language designers about the expressive power of those common features, and to simplify implementations and metatheory of feature-rich languages with polymorphism and subtyping.

1 Introduction

Intersection types [50, 22] and *parametric polymorphism* are common features in many newer mainstream Object-Oriented (OO) languages. Among others intersection types are useful to express *multiple interface inheritance* [21]. They feature in programming languages like Scala [44], TypeScript [40], Ceylon [51] and Flow [31]. These languages also incorporate a form of *parametric polymorphism*, typically generalized to account for subtyping and supporting *bounded quantification* [12]. As programmers get more experienced with the combination of intersection types and polymorphism, they discover new applications. For example, the documentation of TypeScript [41] shows how the two features can express a composition operator for objects that enables an expressive form of statically typed *dynamic inheritance* [20, 32] and *mixin composition* [8]:

```
function extend<A, B>(first: A, second: B): A & B
```

The *polymorphic* function `extend` takes two objects and produces a result whose type is the intersection of the types of the original objects. The implementation of `extend` relies on low level features of JavaScript and is right-biased: the fields or properties of `second` are chosen in favor of the ones in `first`. For example, we can create a new object `jim` as follows:

```
var jim = extend(new Person('Jim'), new ConsoleLogger());
```

© Ningning Xie,Xuan Bi, Bruno C. d. S. Oliveira and Tom Schrijvers;
licensed under Creative Commons License CC-BY

The `jim` object has type `Person & ConsoleLogger`, and acts both as a person and as a console logger. Using `extend` to compose objects is much more flexible than the *static inheritance* mechanisms of common OO languages like Java or Scala. It allows type-checking flexible OO patterns that have been used for many years in many dynamically-typed languages.

Unfortunately, the `extend` function in TypeScript suffers from *ambiguity* issues, and worse, it is not type-safe [2]. Indeed, given two objects with the same field or method names, `extend` does not detect potential conflicts. Instead it silently composes the two objects, using the implementation based on a biased choice. This does implement a mixin semantics, but it has the drawback that it can unintentionally override methods, without any warnings or errors. Additionally, the `extend` function is *not type-safe*: if two objects have the same property name with different types, `extend` may lookup the property of the wrong type.

In the literature of intersection types, `extend` is essentially what has been identified as the *merge operator* [53]. As illustrated by Dunfield [28], the expressive power of the merge operator allows encoding diverse programming language features, promising an economy of theory and implementation. Calculi with *disjoint intersection types* [46, 7, 2] incorporate a *coherent* merge operator. In such calculi the merge operator can merge two terms with *arbitrary* types as long as their types are disjoint; disjointness conflicts are reported as type-errors. Some calculi with disjoint intersection types, such as F_i^+ [7], also support *disjoint polymorphism* [2], which extends System F style universal quantification with a *disjointness constraint.* With disjoint polymorphism we can model `extend` as:

```
let extend A (B * A) (first : A, second : B) : A & B = first ,, second
```

Unlike the TypeScript definition, which relies on type-unsafe features, the definition above includes the full implementation. The definition of `extend` uses the merge operator (`,,`) to compose the two objects. The type variable `B` has a disjointness constraint (`B * A`) which states that `B` must be disjoint from `A`. Disjointness retains the flexibility to encode highly dynamic forms of inheritance, while ensuring both type-safety and the absence of conflicts.

Row polymorphism and disjoint polymorphism Disjoint polymorphism looks quite close to certain forms of *row polymorphism.* Indeed, when restricted to *record types*, row polymorphism with *constrained quantification* [34] provides an approach to recovering an unambiguous semantics for `extend` as well. Constrained quantification extends System F style universal quantification with a *compatibility* constraint. By requiring `B` to be *compatible* with `A`, we can encode a row polymorphic variant of `extend` as:

```
let extend A (B # A) (first : A, second : B) : A || B = first || second
```

Here `A` and `B` are *row variables* standing for *record types*, and `B` is compatible with `A` (`B # A`), which ensures the absence of conflicts. The `||` operator concatenates two records at both the term level and the type level. The key difference between the two implementations of `extend` is that in the version with row variables, `A` and `B` can only stand for record types. In contrast in the version with disjoint polymorphism, `A` and `B` are arbitrary types. In languages with nominal type systems, allowing arbitrary types is important to deal with nominal types of classes, for instance. The encoding of `extend` suggests that at least some functionality of row polymorphism can be captured in a calculus with disjoint polymorphism. Indeed, there are clear analogies between the two mechanisms: the merge operators (`,,` and `||`) are clearly similar; *compatibility* plays a similar role to *disjointness*; and intersection types generalize record type concatenation.

Bounded quantification and disjoint polymorphism Polymorphic object-oriented languages also typically feature *bounded quantification*, which addresses the interaction between polymorphism and subtyping. Bounded quantification generalizes universal quantification by allowing programmers to specify upper bounds on type variables. For example:

```
let getName (A <: Person) (o : A) : (String,A) = (o.name,o)
```

expresses a function `getName` that takes an object `o` whose type is a subtype of `Person`, extracts its name and returns a copy of the object. Note that bounded quantification is useful to avoid the *loss of information problem* of subtyping [11]. Using the simpler type:

```
let getName_bad (o : Person) : (String,Person) = (o.name,o)
```

would lose static type information when given a *subtype* of `Person` as an argument.

An alternative version of `getName` that also does not lose type information is:

```
let getName A (o : A & Person) : (String,A & Person) = (o.name,o)
```

Here, the type variable `A` is unrestricted and represents the statically unknown part of the type of the object. The intersection type `A & Person` ensures that the object must at least contain all properties of `Person`, but does not forget about the statically unknown components. The two versions of `getName` show a common use case in OOP, but they use different features: the first uses *bounded quantification*, while the second uses a combination of intersection types and polymorphism. The connection between bounded quantification and polymorphic intersection types has been informally observed by Pierce [47].

Disjoint polymorphism, *intersection types*, *row polymorphism* and *bounded quantification* provide a range of functionalities for OOP languages. Therefore a language designer may be tempted to design a core language that combines all of these concepts. However, supporting all of them would lead to a significant implementation effort and a complex metatheory with non-trivial interactions between features. Furthermore, a common principle for (core) languages is to avoid overlapping features, which provide different ways to solve the same problem. As we have seen, there seems to be a significant overlap between these features, which goes against that principle.

This paper builds on the similarities between the mechanisms, and shows that forms of both row polymorphism and bounded polymorphism can be recovered by type-safe elaborations into languages with disjoint polymorphism. This result suggests that core languages wishing to support all those features only need to support disjoint polymorphism natively, promising an economy of both the theory and implementation of those languages. To establish the relationship between row, bounded and disjoint polymorphism in a rigorous and precise manner, we formalize elaborations from $\lambda^{||}$ [34], a System F like calculus with row polymorphism, and from kernel $\mathsf{F}_{<:}$ [12], into F_i^+. Our work serves as a guideline for language designers wishing to combine disjoint polymorphism, with bounded quantification and/or row polymorphism. The elaborations are useful to understand exactly what can and cannot be encoded, and to uncover and overcome difficulties. To our surprise, a full encoding of $\lambda^{||}$ is quite subtle: there are subtle differences between compatibility and disjointness. Moreover, certain general forms of bounded quantification are problematic, but all programs in kernel $\mathsf{F}_{<:}$ (the most widely used and decidable fragment of $\mathsf{F}_{<:}$) are encodable.

We make the following specific contributions:

- **A formal elaboration from row to disjoint polymorphism:** We present a formal elaboration from $\lambda^{||}$ to F_i^+ (Section 4). We first identify an intuitive elaboration (Section 4.3). Due to discrepancies between compatibility and disjointness this elaboration does not work for all $\lambda^{||}$ programs. However it is possible to find a simple restriction on $\lambda^{||}$ that allows for the intuitive elaboration to work. We then present a complete, but *non-trivial* elaboration that targets the original $\lambda^{||}$ without restrictions (Section 4.4). While the design space of row polymorphic calculi is very diverse, features in $\lambda^{||}$ are representative of most other calculi. We discuss elaborating other row calculi in Section 6.1.

- **A formal elaboration from bounded to disjoint polymorphism:** We identify a fragment of $\mathsf{F}_{<:}$ that is encodable in terms of polymorphic intersection type systems, by providing an elaboration from *kernel* $\mathsf{F}_{<:}$ to F_i^+ (Section 5). Our elaboration, for the first time, validates the informal observation between polymorphic intersection systems and bounded quantification. We discuss other variants of $\mathsf{F}_{<:}$ in Section 6.2.
- **A discussion of the extra expressive power of disjoint polymorphism:** We identify and discuss specific features of disjoint polymorphism that cannot be easily encoded in $\mathsf{F}_{<:}$ and $\lambda^{||}$ (Section 2.4), including distributivity of intersections over other constructs, and the combination of subtyping and row polymorphism. We discuss other variants of intersection type systems in Section 6.3.
- **Coq formalization:** All elaborations and metatheory of this paper, except for some manual proof for simulation, has been mechanically formalized in the Coq proof assistant, including *type-safety* and *coherence*. The Coq formalization amounts to *18,855* lines of proofs and code (not including blank lines, comments and existing metatheory for F_i^+).[1]

2 Overview

This section introduces the key ideas of the encodings for bounded quantification and row polymorphism. We also discuss the added extra power of disjoint polymorphism over bounded quantification and row polymorphism.

2.1 Background: Disjoint Polymorphism

Disjoint polymorphism [2, 7] combines disjoint intersection types with parametric polymorphism. In particular, F_i^+ [7] supports *intersection types* $A \,\&\, B$ for terms that are both of type A and of type B. With the *merge operator* we can construct terms of an intersection type, like $1 \,,, \mathsf{True}$ of type $\mathsf{Int} \,\&\, \mathsf{Bool}$. Thanks to *subtyping*, a term of type $\mathsf{Int} \,\&\, \mathsf{Bool}$ can also be used as if it had type Int, or as if it had type Bool. F_i^+ requires the two components of a merge to have disjoint types, e.g., $1 \,,, 2 : \mathsf{Int} \,\&\, \mathsf{Int}$ is not allowed, because it is ambiguous which value should be used at type Int. With *disjoint quantification*, it is possible to enforce disjointness when the types of one or both components of a merge contain type variables. For instance, the term $\Lambda(\alpha * \mathsf{Int}).\, \lambda(x : \alpha).\, x \,,, 1$ has type $\forall(\alpha * \mathsf{Int}).\, \alpha \to \alpha \,\&\, \mathsf{Int}$. The disjointness annotation $\alpha * \mathsf{Int}$ allows α to be instantiated only to types that are disjoint from Int. Without a disjointness constraint, the term $\Lambda\alpha.\, \lambda(x : \alpha).\, x \,,, 1$ is rejected. Otherwise such a term would allow α to be instantiated to Int, and thus the function could be applied to the number 2 for example, leading to the *ambiguous* merge $2 \,,, 1$.

2.2 Row Polymorphism through Disjoint Polymorphism

Row types, originally introduced by Wand [60] to model inheritance, provide an approach to typing extensible records. Row types have been studied extensively [35, 11, 52, 42] and have been applied to provide extensibility in various type systems [37, 36, 38]. According to Rémy [52], record calculi can be divided into those that support *free* extension, and those that support *strict* extension. The former allows extension with fields that already exist, whereas the latter does not. In this paper we focus on $\lambda^{||}$, a calculus proposed by Harper and Pierce [34] that extends System F with row polymorphism. $\lambda^{||}$ belongs to the strict

[1] **Note for reviewers:** The Appendix and all Coq proofs can be found in the supplementary materials.

camp and avoids concatenating records with a field label in common by means of *constrained quantification*. A constrained quantifier attaches a constraint list to a type variable, which restricts the instantiations of that type variable to be record types with field labels that are distinct from all the record types in the constraint list. What sets $\lambda^{||}$ apart from other strict record calculi is its ability to merge records with statically unknown fields, and a mechanism to ensure that the resulting record is conflict-free (i.e., no duplicate labels). The following function concatenates two records by the *merge* operator $||$:

$$\mathsf{mergeRcd} = \Lambda(\alpha_1 \,\#\, \mathsf{Empty}).\, \Lambda(\alpha_2 \,\#\, \alpha_1).\, \lambda(x_1 : \alpha_1).\, \lambda(x_2 : \alpha_2).\, x_1 \mid\mid x_2$$

which takes two type variables, each of which *lacks* (#) the appropriate fields (Empty means no constraints at all). The function above can take any record type as its first argument, but the second type must be *compatible* with the first ($\alpha_2 \# \alpha_1$), i.e., the second record cannot have any labels that also occur in the first. These constraints ensure that the resulting record $x_1 \mid\mid x_2$ has no duplicate labels. If later we want to say that the first record x_1 has *at least* a field l_1 of type Int, we can refine the constraint list of α_1, α_2 and the type of x_1 accordingly:

$$\Lambda(\alpha_1 \,\#\, \{l_1 : \mathsf{Int}\}).\, \Lambda(\alpha_2 \,\#\, (\alpha_1, \{l_1 : \mathsf{Int}\})).\, \lambda(x_1 : \alpha_1 \mid\mid \{l_1 : \mathsf{Int}\}).\, \lambda(x_2 : \alpha_2).\, x_1 \mid\mid x_2$$

Encoding with disjoint polymorphism Our encoding of $\lambda^{||}$ into F_i^+ is based on the similarities between the two calculi that the astute reader may have already observed. Indeed, the constrained quantification of record type variables $\Lambda(\alpha \,\#\, R).\, \varepsilon$ is quite similar to the disjoint quantification $\Lambda(\alpha * A).\, E$. They both constrain the use of respectively the record concatenation operator $x_1 \mid\mid x_2$ and the merge operator $x_1 \,,,\, x_2$. Exploiting these similarities, we can encode mergeRcd as follows in F_i^+:

$$\mathsf{mergeAny} = \Lambda(\alpha_1 * \top).\, \Lambda(\alpha_2 * \alpha_1).\, \lambda(x_1 : \alpha_1).\, \lambda(x_2 : \alpha_2).\, x_1 \,,,\, x_2$$

An important difference is that in mergeRcd, α_1 and α_2 are *row variables*: they can only be instantiated with record types. In contrast in mergeAny, α_1 and α_2 are type variables and they can be instantiated with any types, including types which are not records (such as Int).

Formal elaboration To establish the validity of the encoding, we have formalized two different elaborations of $\lambda^{||}$ into F_i^+. The first elaboration exploits the obvious similarity between the two mechanisms. While it clearly works for many example programs, the formalization of the metatheory reveals that the straightforward elaboration does not work for all programs. Indeed, it turns out that there is a subtle difference in the interpretation of the constrained quantification and the disjoint quantification that makes the elaboration break down in some cases. For instance, the $\lambda^{||}$ binder $\Lambda\alpha\#\{l : \mathsf{Int}\}$ expresses that α cannot have the label l *at all*. In contrast, the F_i^+ binder $\Lambda\beta * \{l : \mathsf{Int}\}$ expresses that β cannot have a field l of type Int, but it can have a field l of some other *disjoint* type, say Bool. In what we consider to be contrived programs, this subtle difference invalidates the elaboration. We can eliminate this source of semantic difference by slightly restricting $\lambda^{||}$, which is what we do in the first elaboration. However, in order to handle those contrived (but well-typed) unrestricted $\lambda^{||}$ programs as well, we also present a more complex elaboration that faithfully captures the semantics of constrained quantification in unrestricted $\lambda^{||}$.

2.3 Bounded Quantification through Disjoint Polymorphism

Bounded quantification is a language feature that integrates parametric polymorphism with subtyping. It was first introduced in the language Fun [12] as a means of typing functions that operate uniformly over all subtypes of a given type, and has been the subject of much theoretical and practical effort [9, 47, 48, 39, 13, 11, 18, 25, 49]. In this paper, we focus on System $\mathsf{F}_{<:}$, which is a calculus with bounded quantification that extends System F.

As an illustration of bounded quantification, consider the following definition:

$$f = \lambda(x : \{\mathsf{val} : \mathsf{Int}\}).\,\{\mathsf{orig} = x, \mathsf{val} = x.\mathsf{val} + 1\}$$

The function f has type $\{\mathsf{val} : \mathsf{Int}\} \to \{\mathsf{orig} : \{\mathsf{val} : \mathsf{Int}\}, \mathsf{val} : \mathsf{Int}\}$, but it actually works for all records that have a val field of type Int. Thanks to bounded quantification we can formulate a variant of f that admits this:

$$\mathit{fpoly} = \Lambda(\alpha <: \{\mathsf{val} : \mathsf{Int}\}).\,\lambda(x : \alpha).\,\{\mathsf{orig} = x, \mathsf{val} = x.\mathsf{val} + 1\}$$

The term *fpoly* has type $\forall(\alpha <: \{\mathsf{val} : \mathsf{Int}\}).\,\alpha \to \{\mathsf{orig} : \alpha, \mathsf{val} : \mathsf{Int}\}$. Here the (upper-)bound $\{\mathsf{val} : \mathsf{Int}\}$ restricts the instantiation of the quantified type variable α to subtypes of $\{\mathsf{val} : \mathsf{Int}\}$.

Encoding with disjoint polymorphism Pierce [47] informally discussed an encoding of bounded quantification in terms of intersection types, by reading a bounded quantifier as an abbreviation for an unbounded one with a slightly modified body:

$$\forall(\alpha <: A).\,B \triangleq \forall\beta.\,([\beta \,\&\, A/\alpha]B)$$

For the above example, we have

$$\forall\beta.\,\beta \,\&\, \{\mathsf{val} : \mathsf{Int}\} \to \{\mathsf{orig} : \beta \,\&\, \{\mathsf{val} : \mathsf{Int}\}, \mathsf{val} : \mathsf{Int}\}$$

However, there is no formalization of this encoding, and it is not clear at all what fragment of programs can be encoded. Pierce showed that this is not an encoding for full $\mathsf{F}_{<:}$ as it does not respect the subtyping rule for universal quantification. Nevertheless, after some experimentation, where the encoding was *manually* applied to complex examples, he came to the conclusion that *"the encoding trick works better than might be expected"*. Castagna and Xu [19] even claim that *"bounded quantification does not require any modification"* in their intersection type system due to this encoding. However, due to Pierce's counterexamples, without further qualification, this statement cannot be fully justified.

What is missing is to clarify precisely the expressiveness of this encoding with a type-theoretic formalization. Our work serves as a basis to fill the gaps, by identifying an encodable fragment of $\mathsf{F}_{<:}$, i.e., kernel $\mathsf{F}_{<:}$, and thus, for the first time, validates the informal observation of this encoding.

Formal elaboration We formalize Pierce's informal encoding idea and turn it into a structurally recursive procedure that systematically and simultaneously replaces all bounded quantifiers in a term. While doing this we faced several technical challenges. The first one was the misalignment between the $\mathsf{F}_{<:}$ and F_i^+ type systems: the former is undirected and the latter is bidirectional. This is a source of complication. In particular, we need to add explicit type annotations for all terms whose type cannot be synthesized, but only checked. Another challenge was the implicit use of subsumption in the typing of $\mathsf{F}_{<:}$ terms. We shift around the position in the term where subsumption happens and still arrive at the same type for the whole term. While the different typing derivations may lead to different F_i^+ elaborations, we do not want those different elaborations to have a different meaning. Hence, we must show that the elaboration is *coherent*. Finally we had to identify the class of $\mathsf{F}_{<:}$ programs for which the encoding actually works. This was not clear from the individual examples that Pierce gave, but it was necessary to make a formal statement that characterizes the extent and thus the usefulness of the encoding. Our translation shows that all well-typed kernel $\mathsf{F}_{<:}$ programs are encodable as well-typed F_i^+ programs. We believe that this justifies Pierce's claim that the encoding might work better than expected, as the kernel $\mathsf{F}_{<:}$ fragment is quite important in practice. It is the most common decidable fragment of $\mathsf{F}_{<:}$ and widely used to model key aspects of OO programs.

2.4 The Extra Power of Disjoint Polymorphism

This section identifies some of the additional expressive power of F_i^+ over $\mathsf{F}_{<:}$ and $\lambda^{||}$ alone.

Distributivity, Nested Composition and Family Polymorphism F_i^+ is based on BCD subtyping [4], which features *distributive* subtyping rules, and enables *nested composition* of merges. Nested composition has several applications. In particular it is a key feature to enable *family polymorphism* [29].

With nested composition we can model a combinator that is useful to compose interpretations of *embedded DSLs*. A minimal example [7] is:

```
type R[e] = {lit : Int → e, neg : e → e} -- literal and negative expressions
compose = Λ(a * ⊤). Λ(b * a). λ(r1 : R[a]). λ(r2 : R[b]). (r1 ,, r2) : R[a & b]
```

Here `R[e]` stands for the abstract syntax of a tiny form of arithmetic expressions. The combinator `compose` allows the composition of two arbitrary interpretations (such as evaluation and pretty printing), into a single interpretation that runs both interpretations at once. In F_i^+ this functionality is achieved by simply merging `r1` and `r2`. Nested composition takes care of the details, by implicitly using a form of type-directed code generation, which is triggered by the upcast: `R[a] & R[b] <: R[a & b]` in expression `r1 ,, r2`. The type of `r1 ,, r2` is `R[a] & R[b]`. In F_i^+, due to the distributivity properties of intersections, such a type is a subtype of `R[a & b]`. Importantly, the fact that records are not treated specially in the type language is a key to allowing distributivity, which in turn enables nested composition.

The interested reader can see the work by Bi et al. [6, 7] for more complete examples. These examples illustrate how nested composition provides a simple and elegant solution to the *Expression Problem* (EP) [59]. In essence the approach mimics Ernst's solution to the EP with family polymorphism [30] (which also relies on a form of nested composition).

With bounded quantification alone, `compose` is essentially not expressible. A solution with row polymorphism can be simulated only at the cost of more work:

```
Λ(a # Empty). Λ(b # a). λ(r1 : R[a]). λ(r2 : R[b]).
  { lit = λ(i : Int)    . (r1.lit i      , r2.lit i)
  , neg = λ(e : (a, b)). (r1.neg (fst e), r2.neg (snd e)) }
```

Since row polymorphism does not support nested composition of merges, the code for executing the two interpretations at once has to be explicitly modeled with some tedious boilerplate code. Moreover, the results of the two interpretations have to be stored in a pair, and explicit projections are necessary to access the values.

In essence the manual composition approach employed with row polymorphism is akin to some existing solutions to the EP which need to tediously compose classes in different families manually. For instance, it is well-known that Scala enables solutions to the EP [62]. However, without nested composition those solutions are cluttered with manual composition code. In contrast, solutions based on nested composition are much more concise and elegant thanks to the automatic composition [30, 6, 7].

Subtyping and row typing F_i^+ combines both subtyping and row polymorphism under one roof. The majority of systems with row polymorphism have been employed as an alternative to subtyping (although some row calculi also have subtyping, e.g., [11]). $\lambda^{||}$, in particular, has no subtyping. One argument for row polymorphism is that it also eliminates the *loss of information problem* of subtyping [11]. For example, with subtyping, an identity function:

$$\lambda(x : \{l : \mathsf{Int}\}).\, x$$

with type $\{l : \mathsf{Int}\} \to \{l : \mathsf{Int}\}$ may, inadvertently, lose some precision on the output type. For instance, the function can be applied to the record $\{l = 1, l' = \mathsf{True}\}$, but the result type of such an application is $\{l : \mathsf{Int}\}$ and not $\{l : \mathsf{Int}, l' : \mathsf{Bool}\}$.

$\lambda^{||}$ solves the loss of information problem by formulating such an identity function in a different way:

$$\Lambda(\alpha \# \{l : \mathsf{Int}\}).\, \lambda(x : \{l : \mathsf{Int}\} \mid\mid \alpha).\, x$$

In this function the row variable α stands for any record without a label l. The type of x expresses that x includes a label l, as well as any labels in α. In this function the output type is $\{l : \mathsf{Int}\} \mid\mid \alpha$ as well. Therefore the application of the function to $\{l = 1, l' = \mathsf{True}\}$ has the type $\{l : \mathsf{Int}, l' : \mathsf{Bool}\}$, which does not lose precision.

With disjoint polymorphism we can easily translate the $\lambda^{||}$ approach and reap its benefits too:

$$\Lambda(\alpha * \{l : \mathsf{Int}\}).\, \lambda(x : \{l : \mathsf{Int}\} \,\&\, \alpha).\, x$$

This function, like the row polymorphic version, preserves the precision of the output type.

Nevertheless, for many functions subtyping does not lose precision. For example, the function:

$$\lambda(x : \{l : \mathsf{Int}\}).\, x.l + 1$$

has type $\{l : \mathsf{Int}\} \to \mathsf{Int}$. In this case no matter which record is passed as an argument the output type is as precise as it can be. Note that this function is valid in F_i^+ and, because of subtyping, the record $\{l = 1, l' = \mathsf{True}\}$ is a valid argument. However in $\lambda^{||}$, the only way to allow records with more labels, is to generalize the function to:

$$\Lambda(\alpha \# \{l : \mathsf{Int}\}).\, \lambda(x : \{l : \mathsf{Int}\} \mid\mid \alpha).\, x.l + 1$$

In this case the generalization does not gain any precision, and in fact it requires a more complex type than the version with subtyping.

In summary, unlike $\lambda^{||}$, many functions in F_i^+ can have a simpler non-polymorphic type and still allow for larger records to be used as inputs.

3 Disjoint Polymorphism

This section reviews F_i^+, which serves as target of our elaborations of row and bounded polymorphism. The F_i^+ calculus and its metatheory have been studied already in Bi et al. [7]. We refer to [7] for further details regarding F_i^+'s formalization and metatheory.

3.1 Syntax and Semantics

Syntax The syntax of F_i^+ is given at the top of Figure 1. Types A, B, C include integers Int, the top type $\top$, the bottom type $\bot$, arrows $A \to B$, intersection types $A \,\&\, B$, singleton record types $\{l : A\}$, type variables α and disjoint quantification $\forall(\alpha * A).\, B$. Expressions E include term variables x, integers i, the top value $\top$, abstractions $\lambda x.\, E$, applications $E_1\, E_2$, merge expressions $E_1 \,,, E_2$, annotated terms $E : A$, singleton records $\{l = E\}$, record projections $E.l$, type abstractions $\Lambda(\alpha * A).\, E$ and type applications $E\, A$. Term contexts Γ record types of term variables, and type contexts Δ record disjointness constraints of type variables. Well-formedness of a type or a context are standard and omitted here.

Subtyping The subtyping relation of F_i^+ is presented in the middle of Figure 1. Most rules are standard. For functions (rule S-ARR) and disjoint quantifications (rule S-FORALL), subtyping is covariant in positive positions, and contravariant in negative positions. Rules S-ANDL, S-ANDR, and S-AND for intersection types axiomatize that $A \,\&\, B$ is the greatest lower bound of A and B. Moreover, F_i^+ features BCD-style subtyping [4], where intersections are distributive over other type constructs. Concretely, intersections distribute over arrows (rule S-DISTARR), records (rule S-DISTRCD) and disjoint quantifications (rule S-DISTALL). Rules S-TOPARR, S-TOPRCD, and S-TOPALL are special cases of the distributivity rules, when viewing $\top$ as a 0-ary intersection.

Types	A, B, C	$::=$	$\mathsf{Int} \mid \top \mid \bot \mid A \to B \mid A \,\&\, B \mid \{l : A\} \mid \alpha \mid \forall(\alpha * A).\, B$
Expressions	E	$::=$	$x \mid i \mid \top \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 \,,,\, E_2 \mid E : A \mid \{l = E\} \mid E.l$
		$\mid$	$\Lambda(\alpha * A).\, E \mid E\, A$
Term contexts	Γ	$::=$	$\bullet \mid \Gamma, x : A$
Type contexts	Δ	$::=$	$\bullet \mid \Delta, \alpha * A$

$\boxed{A <: B}$ *(Declarative subtyping)*

$$\textsc{S-Refl}\ \frac{}{A <: A} \qquad \textsc{S-Trans}\ \frac{A_2 <: A_3 \qquad A_1 <: A_2}{A_1 <: A_3} \qquad \textsc{S-Top}\ \frac{}{A <: \top} \qquad \textsc{S-Bot}\ \frac{}{\bot <: A} \qquad \textsc{S-Rcd}\ \frac{A <: B}{\{l : A\} <: \{l : B\}}$$

$$\textsc{S-Arr}\ \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad \textsc{S-Forall}\ \frac{B_1 <: B_2 \qquad A_2 <: A_1}{\forall(\alpha * A_1).\, B_1 <: \forall(\alpha * A_2).\, B_2} \qquad \textsc{S-And}\ \frac{A_1 <: A_2 \qquad A_1 <: A_3}{A_1 <: A_2 \,\&\, A_3}$$

$$\textsc{S-AndL}\ \frac{}{A_1 \,\&\, A_2 <: A_1} \qquad \textsc{S-AndR}\ \frac{}{A_1 \,\&\, A_2 <: A_2} \qquad \textsc{S-DistArr}\ \frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3}$$

$$\textsc{S-DistRcd}\ \frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\}} \qquad \textsc{S-DistAll}\ \frac{}{(\forall(\alpha * A).\, B_1) \,\&\, (\forall(\alpha * A).\, B_2) <: \forall(\alpha * A).\, B_1 \,\&\, B_2}$$

$$\textsc{S-TopArr}\ \frac{}{\top <: \top \to \top} \qquad \textsc{S-TopRcd}\ \frac{}{\top <: \{l : \top\}} \qquad \textsc{S-TopAll}\ \frac{}{\top <: \forall(\alpha * \top).\, \top}$$

$\boxed{\Delta; \Gamma \vdash E \Rightarrow A}$ *(Inference)*

$$\textsc{T-Top}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \top \Rightarrow \top} \qquad \textsc{T-Nat}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{T-Var}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad (x : A) \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A}$$

$$\textsc{T-App}\ \frac{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \qquad \Delta; \Gamma \vdash E_2 \Leftarrow A_1}{\Delta; \Gamma \vdash E_1\, E_2 \Rightarrow A_2} \qquad \textsc{T-TAbs}\ \frac{\Delta \vdash A \qquad \Delta, \alpha * A; \Gamma \vdash E \Rightarrow B}{\Delta; \Gamma \vdash \Lambda(\alpha * A).\, E \Rightarrow \forall(\alpha * A).\, B}$$

$$\textsc{T-Merge}\ \frac{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \qquad \Delta; \Gamma \vdash E_2 \Rightarrow A_2 \qquad \Delta \vdash A_1 * A_2}{\Delta; \Gamma \vdash E_1 \,,,\, E_2 \Rightarrow A_1 \,\&\, A_2} \qquad \textsc{T-Rcd}\ \frac{\Delta; \Gamma \vdash E \Rightarrow A}{\Delta; \Gamma \vdash \{l = E\} \Rightarrow \{l : A\}}$$

$$\textsc{T-Proj}\ \frac{\Delta; \Gamma \vdash E \Rightarrow \{l : A\}}{\Delta; \Gamma \vdash E.l \Rightarrow A} \qquad \textsc{T-Anno}\ \frac{\Delta; \Gamma \vdash E \Leftarrow A}{\Delta; \Gamma \vdash E : A \Rightarrow A} \qquad \textsc{T-TApp}\ \frac{\Delta; \Gamma \vdash E \Rightarrow \forall(\alpha * B).\, C \qquad \Delta \vdash A * B}{\Delta; \Gamma \vdash E\, A \Rightarrow [A/\alpha]C}$$

$\boxed{\Delta; \Gamma \vdash E \Leftarrow A}$ *(Checking)*

$$\textsc{T-Abs}\ \frac{\Delta \vdash A \qquad \Delta; \Gamma, x : A \vdash E \Leftarrow B}{\Delta; \Gamma \vdash \lambda x.\, E \Leftarrow A \to B} \qquad \textsc{T-Sub}\ \frac{\Delta; \Gamma \vdash E \Rightarrow B \qquad B <: A}{\Delta; \Gamma \vdash E \Leftarrow A}$$

Figure 1 Syntax, declarative subtyping, and bidirectional type system of F_i^+.

Typing The bidirectional typing rules for F_i^+ are given at the bottom of Figure 1. The inference judgment $\Delta; \Gamma \vdash E \Rightarrow A$ says that under the type context Δ and the term context Γ, we can synthesize the type A for the expression E. The checking judgment $\Delta; \Gamma \vdash E \Leftarrow A$ checks E against the type A under the contexts Δ and Γ. Most of the typing rules are standard. Rule T-merge says that the merge expression $E_1 \,,,\, E_2$ is well-typed if both

$\boxed{\rceil A \lceil}$ *(Top-like types)*

$$\textsc{TL-top}\ \frac{}{\rceil \top \lceil} \qquad \textsc{TL-and}\ \frac{\rceil A \lceil \quad \rceil B \lceil}{\rceil A \,\&\, B \lceil} \qquad \textsc{TL-arr}\ \frac{\rceil B \lceil}{\rceil A \to B \lceil} \qquad \textsc{TL-rcd}\ \frac{\rceil A \lceil}{\rceil \{l : A\} \lceil} \qquad \textsc{TL-all}\ \frac{\rceil B \lceil}{\rceil \forall (\alpha * A).\, B \lceil}$$

$\boxed{\Delta \vdash A * B}$ *(Disjointness)*

$$\textsc{D-topL}\ \frac{\rceil A \lceil}{\Delta \vdash A * B} \qquad \textsc{D-topR}\ \frac{\rceil B \lceil}{\Delta \vdash A * B} \qquad \textsc{D-ax}\ \frac{A *_{ax} B}{\Delta \vdash A * B} \qquad \textsc{D-arr}\ \frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2}$$

$$\textsc{D-andL}\ \frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B} \qquad \textsc{D-andR}\ \frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2} \qquad \textsc{D-rcdNeq}\ \frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}}$$

$$\textsc{D-rcdEq}\ \frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}} \qquad \textsc{D-tvarL}\ \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B} \qquad \textsc{D-tvarR}\ \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha}$$

$$\textsc{D-forall}\ \frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall (\alpha * A_1).\, B_1 * \forall (\alpha * A_2).\, B_2}$$

Figure 2 Selected rules for disjointness.

sub-expressions are well-typed, and their types are *disjoint*. The disjointness judgment $\Delta \vdash A_1 * A_2$ is important to rule out invalid merges, such as $1 \,,,\, 2$. Rule T-tabs says that, when typing a type abstraction, we put the disjointness constraint into the type context and then type-check the body. Conversely, rule T-tapp checks that the type argument should satisfy the disjointness constraint.

Disjointness Figure 2 presents the rules of the disjointness relation. Essentially, disjointness checks whether the merge of two expressions preserves coherence. Rules D-topL and D-topR say that *top-like* types are disjoint with any type. The top-like predicate $\rceil A \lceil$, given at the top of Figure 2, captures the set of types that are isomorphic to $\top$. Disjointness axioms $A *_{ax} B$ (appearing in rule D-ax) take care of two types with different type constructors (e.g., Int and records). The axiom rules can be found in Appendix A.2. The other disjointness rules are standard and explained in detail in previous work [46, 2]. Finally, we note that subtyping preserves disjointness.

▶ **Lemma 1** (Subtyping preserves disjointness). *If $\Delta \vdash A * B$ and $B <: C$, then $\Delta \vdash A * C$.*

3.2 Elaboration and Coherence

The dynamic semantics of F_i^+ is given by a type-directed elaboration ($\rightsquigarrow e$) into another calculus, F_{co}, a variant of System F with explicit coercions. The full definition of F_{co} and the elaboration process can be found in Appendix B. The main challenge of the elaboration is that, due to the non-deterministic nature of the declarative type system, an F_i^+ expression can elaborate to different F_{co} expressions. For example, the subtyping rules S-and, S-andl, and S-andr overlap with each other when both sides are intersections, leading to different coercions depending on the order in which these rules are applied. To establish coherence for F_i^+, Bi et al. [7] resort to contextual equivalence, and they prove that different elaborations of the same F_i^+ expression are contextually equivalent. More formally, $\Delta; \Gamma \vdash e_1 \backsimeq_{ctx} e_2$ means that two F_{co} expressions are contextually equivalent under the corresponding elaboration contexts of Δ and Γ. We state the central coherence lemma below.

▶ **Theorem 2** (Coherence of F_i^+). *We have that*

- *If* $\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e_1$, *and* $\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e_2$, *then* $\Delta; \Gamma \vdash e_1 \backsimeq_{ctx} e_2$.
- *If* $\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e_1$, *and* $\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e_2$, *then* $\Delta; \Gamma \vdash e_1 \backsimeq_{ctx} e_2$.

4 Encoding Row Polymorphism

This section shows how to systematically elaborate $\lambda^{||}$ [34]—a polymorphic record calculus with *constrained quantification*—into F_i^+. We first identify a simple and direct elaboration for a fragment of $\lambda^{||}$, and then present a carefully crafted elaboration of full $\lambda^{||}$ using a more sophisticated elaboration.

4.1 Syntax of $\lambda^{||}$

We start by briefly reviewing the syntax of $\lambda^{||}$, shown at the top of Figure 3. Metavariable t ranges over types, which include the integer type Int, function types $t_1 \to t_2$, constrained quantifications $\forall \alpha \,\#\, R.\, t$ and record types r. Record types are built from record type variables α, the empty record type Empty, single-field records $\{l : t\}$ and record merges $r_1 \mid\mid r_2$.[2] A constraint list R of record types is used to constrain instantiations of record type variables.

Metavariable ε ranges over terms, including term variables x, integers i, lambda abstractions $\lambda(x : t).\, \varepsilon$, function applications $\varepsilon_1\, \varepsilon_2$, the empty record empty, single-field records $\{l = \varepsilon\}$, record merges $\varepsilon_1 \mid\mid \varepsilon_2$, record restrictions $\varepsilon \backslash l$, record projections $\varepsilon.l$, type abstractions $\Lambda(\alpha \,\#\, R).\, \varepsilon$ and type applications $\varepsilon\ [r]$. As a side note, from the syntax of type applications $\varepsilon\ [r]$, it can already be seen that $\lambda^{||}$ only supports quantification over *record types*.

4.2 Typing Rules of $\lambda^{||}$

The type system of $\lambda^{||}$ consists of several conventional judgments. The complete set of rules appears in Appendix C.2. Figure 3 presents selected well-formedness rules for record types. A merge $r_1 \mid\mid r_2$ is well-formed in context T if r_1 and r_2 are well-formed, and moreover, r_1 and r_2 are compatible in T (rule WFR-MERGE)—the most important judgment in $\lambda^{||}$, as we will explain next.

Compatibility The compatibility relation in the middle of Figure 3 plays a central role in $\lambda^{||}$. It is the underlying mechanism for deciding when merging two records is "sensible". Informally, $T \vdash r_1 \,\#\, r_2$ holds if r_1 lacks every field contained in r_2 and vice versa. Compatibility is symmetric (rule CMP-SYMM) and respects type equivalence (rule CMP-EQ). Rule CMP-BASE says that if a record is compatible with $\{l : t\}$, it is also compatible with every record $\{l : t'\}$ with the same label l. A type variable is compatible with the records in its constraint list (rule CMP-TVAR). Two single-field records are compatible if they have different labels (rule CMP-BASEBASE). The remaining rules are self-explanatory; we refer the reader to [34] for further explanation. The judgment of constraint list satisfaction $T \vdash r \,\#\, R$ ensures that r is compatible with every record in the constraint list R.

[2] The original $\lambda^{||}$ also includes record type restrictions $r \setminus l$, which can be systematically erased using type equivalence, thus we omit type-level restrictions but keep term-level restrictions.

Types	t	$::=$	$\mathsf{Int} \mid t_1 \to t_2 \mid \forall \alpha \# R.\, t \mid r$
Records	r	$::=$	$\alpha \mid \mathsf{Empty} \mid \{l : t\} \mid r_1 \mid\mid r_2$
Constraint lists	R	$::=$	$\diamond \mid r, R$
Terms	ε	$::=$	$x \mid i \mid \lambda(x : t).\, \varepsilon \mid \varepsilon_1\, \varepsilon_2 \mid \mathsf{empty} \mid \{l = \varepsilon\} \mid \varepsilon_1 \mid\mid \varepsilon_2$
		$\mid$	$\varepsilon \setminus l \mid \varepsilon.l \mid \Lambda(\alpha \# R).\, \varepsilon \mid \varepsilon\ [r]$
Term contexts	G	$::=$	$\diamond \mid G, x : t$
Type contexts	T	$::=$	$\diamond \mid T, \alpha \# R$

$\boxed{T \vdash r \ \mathsf{record}}$ *(Well-formed record types)*

$$\textsc{wfr-Var}\ \frac{(\alpha \# R) \in T}{T \vdash \alpha \ \mathsf{record}} \qquad \textsc{wfr-Merge}\ \frac{T \vdash r_1 \ \mathsf{record} \qquad T \vdash r_2 \ \mathsf{record} \qquad T \vdash r_1 \# r_2}{T \vdash r_1 \mid\mid r_2 \ \mathsf{record}}$$

$\boxed{T \vdash r_1 \# r_2}$ *(Compatibility)*

$$\textsc{cmp-Eq}\ \frac{T \vdash r \# s \qquad r \sim r' \qquad s \sim s'}{T \vdash r' \# s'} \qquad \textsc{cmp-Symm}\ \frac{T \vdash r \# s}{T \vdash s \# r} \qquad \textsc{cmp-Base}\ \frac{T \vdash r \# \{l : t\} \qquad T \vdash t' \ \mathsf{type}}{T \vdash r \# \{l : t'\}}$$

$$\textsc{cmp-TVar}\ \frac{(\alpha \# R) \in T \qquad T \vdash R \ \mathsf{ok} \qquad r \in R}{T \vdash \alpha \# r} \qquad \textsc{cmp-MergeE}\ \frac{T \vdash r \# (s_1 \mid\mid s_2)}{T \vdash r \# s_i} \qquad \textsc{cmp-Empty}\ \frac{T \vdash r \ \mathsf{record}}{T \vdash r \# \mathsf{Empty}}$$

$$\textsc{cmp-MergeI}\ \frac{T \vdash s_1 \# s_2 \qquad T \vdash r \# s_1 \qquad T \vdash r \# s_2}{T \vdash r \# (s_1 \mid\mid s_2)} \qquad \textsc{cmp-BaseBase}\ \frac{l \neq l' \qquad T \vdash t \ \mathsf{type} \qquad T \vdash t' \ \mathsf{type}}{T \vdash \{l : t\} \# \{l' : t'\}}$$

$\boxed{T \vdash r \# R}$ *(Constraint list satisfaction)*

$$\textsc{cmpList-Nil}\ \frac{T \vdash r \ \mathsf{record}}{T \vdash r \# \diamond} \qquad \textsc{cmpList-Cons}\ \frac{T \vdash r \# r' \qquad T \vdash r \# R}{T \vdash r \# r', R}$$

$\boxed{t_1 \sim t_2}$ *(Type equivalence)*

$$\textsc{teq-MergeAssoc}\ \frac{}{r_1 \mid\mid (r_2 \mid\mid r_3) \sim (r_1 \mid\mid r_2) \mid\mid r_3} \qquad \textsc{teq-MergeComm}\ \frac{}{r_1 \mid\mid r_2 \sim r_2 \mid\mid r_1} \qquad \textsc{teq-MergeUnit}\ \frac{}{r \mid\mid \mathsf{Empty} \sim r} \qquad \textsc{teq-CongAll}\ \frac{R \sim R' \qquad t \sim t'}{\forall \alpha \# R.\, t \sim \forall \alpha \# R'.\, t'}$$

$\boxed{R_1 \sim R_2}$ *(Constraint list equivalence)*

$$\textsc{ceq-Swap}\ \frac{}{r, (r', R) \sim r', (r, R)} \qquad \textsc{ceq-Merge}\ \frac{}{(r_1 \mid\mid r_2), R \sim r_1, (r_2, R)} \qquad \textsc{ceq-Empty}\ \frac{}{\mathsf{Empty}, R \sim R} \qquad \textsc{ceq-Base}\ \frac{}{\{l : t\}, R \sim \{l : t'\}, R}$$

Figure 3 Syntax, and selected rules of $\lambda^{||}$.

Type equivalence Unlike F_i^+, $\lambda^{||}$ does not have subtyping. Instead, $\lambda^{||}$ uses type equivalence to convert terms of one type to another. A selection of the rules defining equivalence of types and constraint lists appears at the bottom of Figure 3. The relation $t_1 \sim t_2$ is an equivalence relation, and is a congruence with respect to the type constructors. Merge is associative (rule teq-MergeAssoc), commutative (rule teq-MergeComm), and has Empty as its unit

$\boxed{T; G \vdash \varepsilon : t \leadsto E}$ *(Type-directed elaboration)*

$$\textsc{wtt-Eq}\quad \frac{T; G \vdash \varepsilon : t \leadsto E \qquad T \vdash t' \ \mathsf{type} \qquad t \sim t'}{T; G \vdash \varepsilon : t' \leadsto E : \llbracket t' \rrbracket}$$

$$\textsc{wtt-Base}\quad \frac{T; G \vdash \varepsilon : t \leadsto E}{T; G \vdash \{l = \varepsilon\} : \{l : t\} \leadsto \{l = E\}}$$

$$\textsc{wtt-Restr}\quad \frac{T; G \vdash \varepsilon : \{l : t\} \mathbin{||} r \leadsto E}{T; G \vdash \varepsilon \setminus l : r \leadsto E : \llbracket r \rrbracket}$$

$$\textsc{wtt-Select}\quad \frac{T; G \vdash \varepsilon : \{l : t\} \mathbin{||} r \leadsto E}{T; G \vdash \varepsilon.l : t \leadsto (E : \{l : \llbracket t \rrbracket\}).l}$$

$$\textsc{wtt-Empty}\quad \frac{T \ \mathsf{ok} \qquad T \vdash G \ \mathsf{ok}}{T; G \vdash \mathsf{empty} : \mathsf{Empty} \leadsto \top}$$

$$\textsc{wtt-Merge}\quad \frac{\begin{array}{c} T; G \vdash \varepsilon_1 : r_1 \leadsto E_1 \\ T; G \vdash \varepsilon_2 : r_2 \leadsto E_2 \qquad T \vdash r_1 \# r_2 \end{array}}{T; G \vdash \varepsilon_1 \mathbin{||} \varepsilon_2 : r_1 \mathbin{||} r_2 \leadsto E_1 \,,, \, E_2}$$

$$\textsc{wtt-AllE}\quad \frac{T; G \vdash \varepsilon : \forall \alpha \# R.\, t \leadsto E \qquad T \vdash r \# R}{T; G \vdash \varepsilon\ [r] : [r/\alpha]t \leadsto E \, \llbracket r \rrbracket \, \llbracket r \rrbracket_\perp}$$

$$\textsc{wtt-AllI}\quad \frac{T \vdash R \ \mathsf{ok} \qquad T, \alpha \# R; G \vdash \varepsilon : t \leadsto E}{T; G \vdash \Lambda(\alpha \# R).\, \varepsilon : \forall \alpha \# R.\, t \leadsto \Lambda(\alpha * \llbracket R \rrbracket).\, \Lambda(\alpha_\perp * \llbracket R \rrbracket).\, E}$$

Figure 4 Selected typing rules of $\lambda^{||}$ with elaboration.

(rule TEQ-MERGEUNIT). As a consequence, records are identified up to permutations. The equivalence of constrained quantification (rule TEQ-CONGALL) relies on the equivalence of constraint lists $R_1 \sim R_2$. Again, it is an equivalence relation, and it respects type equivalence. Constraint lists are essentially finite sets, so order is irrelevant (rule CEQ-SWAP). Merges of constraints can be "flattened" (rule CEQ-MERGE), and occurrences of Empty may be eliminated (rule CEQ-EMPTY). The last rule CEQ-BASE is quite interesting: it implies that the types of single-field records are ignored. The reason is that, as far as compatibility is concerned, only labels matter, thus changing the types of records in constraint lists will not affect their compatibility relation. We will have more to say about this in Section 4.3.

Typing rules A selection of typing rules is shown in Figure 4. In a first reading, the gray parts can be ignored. Most of the typing rules are quite standard. Typing is invariant under type equivalence (rule WTT-EQ). Two terms can be merged if their types are compatible (rule WTT-MERGE). Type application $\varepsilon\ [r]$ is well-typed if the type argument r satisfies the constraints R (rule WTT-ALLE).

▶ Remark 3. We have made a few simplifications compared to the original $\lambda^{||}$, notably the typing of record selection (rule WTT-SELECT) and restriction (rule WTT-RESTR). In the original formulation, both typing rules use a partial function r_l that denotes the type associated with label l in r. Instead of using partial functions, here we explicitly expose the expected label in a record. It can be shown that if label l is present in record type r, then the fields in r can be rearranged so that l comes first by type equivalence. This formulation was also adopted by Leijen [35].

4.3 A Simple yet Incomplete Encoding

The similarities between $\lambda^{||}$ and F_i^+, which the astute reader may have already observed, suggest an intuitive elaboration scheme. On the syntactic level, it is easy to see a one-to-one correspondence between $\lambda^{||}$ types and F_i^+ types. We use $\llbracket t \rrbracket$ to denote the elaboration function from $\lambda^{||}$ types to F_i^+ types, whose formal definition is given at the top of Figure 5.

$$
\begin{array}{rcl|rcl}
\boxed{[\![t]\!]} \quad [\![\mathsf{Int}]\!] & = & \mathsf{Int} & \boxed{[\![R]\!]} \quad [\![\diamond]\!] & = & \top \\
[\![t_1 \to t_2]\!] & = & [\![t_1]\!] \to [\![t_2]\!] & [\![r, R]\!] & = & [\![r]\!] \,\&\, [\![R]\!] \\
[\![\forall \alpha \,\#\, R.\, t]\!] & = & \forall (\alpha * [\![R]\!]).\, [\![t]\!] & \boxed{[\![T]\!]} \quad [\![\diamond]\!] & = & \bullet \\
[\![\alpha]\!] & = & \alpha & [\![T, \alpha \,\#\, R]\!] & = & [\![T]\!], \alpha * [\![R]\!] \\
[\![\mathsf{Empty}]\!] & = & \top & \boxed{[\![G]\!]} \quad [\![\diamond]\!] & = & \bullet \\
[\![\{l : t\}]\!] & = & \{l : [\![t]\!]\} & [\![G, x : t]\!] & = & [\![G]\!], x : [\![t]\!] \\
[\![r_1 \mid\mid r_2]\!] & = & [\![r_1]\!] \,\&\, [\![r_2]\!] & & &
\end{array}
$$

$\boxed{T; G \vdash \varepsilon : t \rightsquigarrow_i E}$ *(Type-directed elaboration)*

$$
\textsc{wtti-Eq}\quad \frac{T; G \vdash \varepsilon : t \rightsquigarrow_i E \qquad T \vdash t' \ \mathsf{type} \qquad t \sim t'}{T; G \vdash \varepsilon : t' \rightsquigarrow_i E : [\![t']\!]}
\qquad
\textsc{wtti-Base}\quad \frac{T; G \vdash \varepsilon : t \rightsquigarrow_i E}{T; G \vdash \{l = \varepsilon\} : \{l : t\} \rightsquigarrow_i \{l = E\}}
$$

$$
\textsc{wtti-AllI}\quad \frac{T \vdash R \ \mathsf{ok} \qquad T, \alpha \,\#\, R; G \vdash \varepsilon : t \rightsquigarrow_i E}{T; G \vdash \Lambda(\alpha \,\#\, R).\, \varepsilon : \forall \alpha \,\#\, R.\, t \rightsquigarrow_i \Lambda(\alpha * [\![R]\!]).\, E}
\qquad
\textsc{wtti-AllE}\quad \frac{T; G \vdash \varepsilon : \forall \alpha \,\#\, R.\, t \rightsquigarrow_i E \qquad T \vdash r \,\#\, R}{T; G \vdash \varepsilon\ [r] : [r/\alpha]t \rightsquigarrow_i E\, [\![r]\!]}
$$

Figure 5 Intuitive elaboration functions, and selected type-directed elaboration from $\lambda^{||}$ to F_i^+.

Elaboration of expressions is also easy. Constrained type abstractions $\Lambda(\alpha \,\#\, R).\, \varepsilon$ correspond to $\Lambda(\alpha * A).\, E$; record merges can be simulated by the more general merge operator of F_i^+; record restriction can be modeled as annotated terms, and so on. On the semantic level, well-formedness judgments of $\lambda^{||}$ match with well-formedness judgments of F_i^+. The compatibility relation corresponds to the disjointness relation. What might not be so obvious is that type equivalence is expressible via subtyping. More specifically, $t_1 \sim t_2$ induces two subtyping relations: $[\![t_1]\!] <: [\![t_2]\!]$ and $[\![t_2]\!] <: [\![t_1]\!]$. Under this elaboration scheme, the full definition of type-directed elaboration, denoted as $T; G \vdash \varepsilon : t \rightsquigarrow_i E$, where i stands for "intuitive", is simple (selected rules are given at the bottom of Figure 5). With all these in mind, let us consider two examples.

▶ **Example 4.** Consider the term $\Lambda(\alpha \,\#\, \{l : \mathsf{Int}\}).\, \lambda(x : \alpha).\, x$. This term can be assigned the type (among others) $\forall \alpha \,\#\, \{l : \mathsf{Int}\}.\, \alpha \to \alpha$, and its F_i^+ counterpart $\Lambda(\alpha * \{l : \mathsf{Int}\}).\, \lambda(x : \alpha).\, x$ has type $\forall(\alpha * \{l : \mathsf{Int}\}).\, \alpha \to \alpha$, which corresponds directly to $\forall \alpha \,\#\, \{l : \mathsf{Int}\}.\, \alpha \to \alpha$. In $\lambda^{||}$, the same term could also be assigned type $\forall \alpha \,\#\, \{l : \mathsf{Bool}\}.\, \alpha \to \alpha$ (rule wtt-Eq), since $\forall \alpha \,\#\, \{l : \mathsf{Bool}\}.\, \alpha \to \alpha$ is equivalent to $\forall \alpha \,\#\, \{l : \mathsf{Int}\}.\, \alpha \to \alpha$ by rules teq-CongAll and ceq-Base. However, in F_i^+, these two types have no relationship at all—$\forall(\alpha * \{l : \mathsf{Int}\}).\, \alpha \to \alpha$ is not the same as $\forall(\alpha * \{l : \mathsf{Bool}\}).\, \alpha \to \alpha$, and indeed it should not be, as these two types have completely different meanings!

▶ **Example 5.** Consider the term $\varepsilon = \Lambda(\alpha \,\#\, \{l : \mathsf{Bool}\}).\, \lambda(x : \alpha).\, \lambda(y : \{l : \mathsf{Int}\}).\, x \mid\mid y$. This term has type $\forall \alpha \,\#\, \{l : \mathsf{Bool}\}.\, \alpha \to \{l : \mathsf{Int}\} \to \alpha \mid\mid \{l : \mathsf{Int}\}$, and its "obvious" elaboration is $E = \Lambda(\alpha * \{l : \mathsf{Bool}\}).\, \lambda(x : \alpha).\, \lambda(y : \{l : \mathsf{Int}\}).\, x \,,,\, y$. However, expression E is ill-typed in F_i^+: we *cannot* merge x with y because their types (α and $\{l : \mathsf{Int}\}$ respectively) are not disjoint. Allowing it to type-check causes incoherence: evaluating $(E\, \{l : \mathsf{Int}\}\, \{l = 1\}\, \{l = 2\}).l$ could result in 1 or 2!

These examples underline a crucial observation: disjointness is more *fine-grained* than compatibility. Unlike F_i^+, the existence of ε in $\lambda^{||}$ will not cause incoherence because compatibility can only relate records with different labels, and thus ε can only be applied to records without label l at all. So $\lambda^{||}$ rejects type application $\varepsilon\ [\{l : \mathsf{Int}\}]$ in the first place. However, disjointness also relates records with the same label as long as their types are disjoint, i.e., rule D-rcdEq. Section 2.4 illustrates the importance of rule D-rcdEq for distributivity, which is not supported by $\lambda^{||}$. A careful comparison between the two calculi reveals that two rules are "to blame": rule ceq-Base and rule cmp-Base, which are the cause for the problem in Example 4 and Example 5 respectively.

$$\frac{}{\{l : t\}, R \sim \{l : t'\}, R}\ \textsc{ceq-Base} \qquad \frac{T \vdash r \# \{l : t\} \qquad T \vdash t' \ \mathsf{type}}{T \vdash r \# \{l : t'\}}\ \textsc{cmp-Base}$$

Yet, both Example 4 and Example 5 seem contrived. From the expression $\Lambda(\alpha \# \{l : \mathsf{Int}\}).\, \lambda(x : \alpha).\, x$, the user can reasonably expect the type to be $\forall \alpha \# \{l : \mathsf{Int}\}.\, \alpha \to \alpha$. For ε, an *equivalent* definition with more sensible and readable annotation is $\varepsilon' = \Lambda(\alpha \# \{l : \mathsf{Int}\}).\, \lambda(x : \alpha).\, \lambda(y : \{l : \mathsf{Int}\}).\, x \mid\mid y$, whose corresponding elaboration type-checks successfully. We believe that programs with the same issue always have some *equivalent* accepted programs by changing some type annotations.

We propose a restricted $\lambda^{||}$ by: (1) replacing rule ceq-Base with rule ceq-BaseAlt; and (2) removing rule cmp-Base. We conjecture that this change has no practical consequences and no expressiveness is lost. Moreover, the restrictions coincide with the observation in Harper and Pierce [34]: *we may normalize constraint lists into the form $l_1, \ldots, l_n, \alpha_1, \ldots, \alpha_m$ where the l_i's are labels and the α_i's are record type variables.* The normalization then validates the change of rules.

$$\frac{t \sim t'}{\{l : t\}, R \sim \{l : t'\}, R}\ \textsc{ceq-BaseAlt}$$

In return, we can prove the intuitive elaboration for restricted $\lambda^{||}$ is, indeed, sound:

▶ **Theorem 6** (Type-safety of $\rightsquigarrow_i$ elaboration). *If $T; G \vdash \varepsilon : t \rightsquigarrow_i E$ then $[\![T]\!]; [\![G]\!] \vdash E \Rightarrow [\![t]\!]$.*

4.4 A Complete Encoding of $\lambda^{||}$ and its Challenges

One criticism to the intuitive encoding is that it does not fully model $\lambda^{||}$: fewer expressions type-check in the modified $\lambda^{||}$. Thus, we present a carefully designed encoding that is able to elaborate the original $\lambda^{||}$ to F_i^+ *without* any restrictions at all. It is highly non-trivial and reveals the essence of constrained quantification from the point of view of disjointness.

First, let us take a step back and have another look at Example 5. As we have discussed, the root cause is rule cmp-Base, which says that *if a record is compatible with a single-field record $\{l : t\}$, then it is compatible with every single-field record $\{l : t'\}$.* To express the essence of rule cmp-Base in F_i^+, we utilize the bottom type $\bot$. In F_i^+, according to Lemma 1, if some type A is disjoint to $\{l : \bot\}$, then, because $\{l : \bot\} <: \{l : B\}$ (by rules S-rcd and S-bot) for any B, we have that A is disjoint to $\{l : B\}$. In other words, in F_i^+, *if a record is disjoint to $\{l : \bot\}$, then it is disjoint to every single-field record $\{l : A\}$.*

▶ **Lemma 7** (Disjointness to records with bottom). *If $\Delta \vdash A * \{l : \bot\}$, then $\Delta \vdash A * \{l : B\}$ for all B.*

Essentially, a compatibility constraint with $\{l : t\}$ in $\lambda^{||}$ corresponds to a disjointness constraint to $\{l : \bot\}$ in F_i^+. Thus, we *bottom-elaborate* the record types that *appear in a constraint list*: if a record $\{l : t\}$ appears in a constraint list, then it is bottom-elaborated to

$\{l : \bot\}$. For Example 4, both $\forall\alpha \# \{l : \mathsf{Int}\}. \alpha \to \alpha$ and $\forall\alpha \# \{l : \mathsf{Bool}\}. \alpha \to \alpha$ elaborate to $\forall(\alpha * \{l : \bot\}). \alpha \to \alpha$. For Example 5, ε elaborates to $E' = \Lambda(\alpha * \{l : \bot\}). \lambda(x : \alpha). \lambda(y : \{l : \mathsf{Int}\}). x \,,, \, y$, which type-checks in F_i^+.

▶ **Example 8.** Now consider the $\lambda^{||}$ term

$$\varepsilon_1 = (\Lambda(\alpha \# \mathsf{Empty}). \lambda(x : (\forall\beta \# \alpha. \mathsf{Int})). 1)\ [\{l : \mathsf{Int}\}]\ (\Lambda(\beta \# \{l : \mathsf{Int}\}). 2)$$

The term type-checks in $\lambda^{||}$ and has type Int. During elaboration, we treat records differently according to where they occur. For the type argument $\{l : \mathsf{Int}\}$, since it is not in a constraint list, we elaborate it normally to $\{l : \mathsf{Int}\}$. For the term argument $(\Lambda(\beta \# \{l : \mathsf{Int}\}). 2)$, since the record $\{l : \mathsf{Int}\}$ appears in a constraint list, we elaborate the term argument to $(\Lambda(\beta * \{l : \bot\}). 2)$. The whole term is then elaborated to

$$E_1 = (\Lambda(\alpha * \top). ((\lambda x. 1) : (\forall(\beta * \alpha). \mathsf{Int}) \to \mathsf{Int}))\ \{l : \mathsf{Int}\}\ (\Lambda(\beta * \{l : \bot\}). 2)$$

However, E_1 fails to type-check in F_i^+: after type application, we substitute α with the type argument $\{l : \mathsf{Int}\}$ in x's type $(\forall(\beta * \alpha). \mathsf{Int})$, yielding $(\forall(\beta * \{l : \mathsf{Int}\}). \mathsf{Int})$, whereas the term argument has type $(\forall(\beta * \{l : \bot\}). \mathsf{Int})$, which does not match (and is not a subtype of) the expected parameter type!

The tricky part here is that, for type variables that appear in the constraint list, after type application, the elaborated disjointness constraint contains the original type argument instead of the bottom-elaborated type. In this case, the result type of type application, i.e., $((\forall(\beta * \{l : \mathsf{Int}\}). \mathsf{Int}) \to \mathsf{Int})$, has $\{l : \mathsf{Int}\}$ instead of $\{l : \bot\}$ in the disjointness constraint.

Apparently we cannot bottom-elaborate every type argument, or otherwise we would lose type information for records. For example, $((\Lambda(\alpha \# \mathsf{Empty}). \lambda(x : \alpha). x)\ [\{l : \mathsf{Int}\}]\ \{l = 1\}).l{+}1$ should not elaborate to $((\Lambda(\alpha * \top). (\lambda x. x) : \alpha \to \alpha)\ \{l : \bot\}\ \{l = 1\}).l + 1$, which is ill-typed.

Therefore, we *bottom-elaborate record variables that appear in a constraint list.* To this end, we map a record type variable α to a pair of type variables α and $\alpha_\bot$, where $\alpha_\bot$ is used in the disjointness constraint. Note that, $\alpha_\bot$ is *not* a new sort of type variable–we can use α_1 or α_2 as well—the subscript $\bot$ here is only for readability. The bottom-elaborated type variable $\alpha_\bot$ is introduced by an extra type abstraction. While α takes the normal type argument, $\alpha_\bot$ takes an extra bottom-elaborated type argument. As an example, the expression ε_1 in Example 8 is elaborated to E_1', which type-checks successfully in F_i^+, where the differences from E_1 are highlighted in gray.

$$E_1' = (\Lambda(\alpha * \top).\ \boxed{\Lambda(\alpha_\bot * \top)}\ . (\lambda x. 1) : (\forall(\beta * \boxed{\alpha_\bot}). \mathsf{Int}) \to \mathsf{Int})\ \{l : \mathsf{Int}\}\ \boxed{\{l : \bot\}}\ (\Lambda(\beta * \{l : \bot\}). 2)$$

Intentionally, $\alpha_\bot$ is a *subtype* of α, as it always takes bottom-elaborated type arguments that are subtype of the original type arguments. For example, $\{l : \bot\}$ is a subtype of $\{l : \mathsf{Int}\}$. However, the type system is unaware of this observation.

▶ **Example 9.** Consider the term

$$\varepsilon_2 = \Lambda(\alpha \# \mathsf{Empty}). \Lambda(\beta \# \alpha). \lambda(x : \alpha). \lambda(y : \beta). x \mid\mid y.$$

Under the current approach, it elaborates to

$$E_2 = \Lambda(\alpha * \top). \Lambda(\alpha_\bot * \top). \Lambda(\beta * \alpha_\bot). \Lambda(\beta_\bot * \alpha_\bot). \lambda(x : \alpha). \lambda(y : \beta). x \,,, \, y$$

However, the merge $x \,,, \, y$ fails to type-check, as we do not have the information that $\alpha * \beta$. We only have $\beta * \alpha_\bot$ in the context. If the system could know that $\alpha_\bot <: \alpha$, then by Lemma 1 we could derive $\beta * \alpha$.

Twisting F_i^+ by adding the axiom $\alpha_\bot <: \alpha$ is unsatisfactory, as it complicates the subtyping relation and also significantly affects the metatheory. Our solution is to include both the

$$
\begin{array}{lrcl}
\boxed{[\![t]\!]} & [\![\mathsf{Int}]\!] & = & \mathsf{Int} \\
 & [\![t_1 \to t_2]\!] & = & [\![t_1]\!] \to [\![t_2]\!] \\
 & [\![\forall \alpha \,\#\, R.\, t]\!] & = & \forall(\alpha * [\![R]\!]).\, \forall(\alpha_\bot * [\![R]\!]).\, [\![t]\!] \\
 & [\![\alpha]\!] & = & \alpha \\
 & [\![\mathsf{Empty}]\!] & = & \top \\
 & [\![\{l : t\}]\!] & = & \{l : [\![t]\!]\} \\
 & [\![r_1 \mid\mid r_2]\!] & = & [\![r_1]\!] \,\&\, [\![r_2]\!] \\
\boxed{[\![G]\!]} & [\![\diamond]\!] & = & \bullet \\
 & [\![G, x : t]\!] & = & [\![G]\!], x : [\![t]\!]
\end{array}
\qquad
\begin{array}{lrcl}
\boxed{[\![r]\!]_\bot} & [\![\alpha]\!]_\bot & = & \alpha_\bot \\
 & [\![\mathsf{Empty}]\!]_\bot & = & \top \\
 & [\![\{l : t\}]\!]_\bot & = & \{l : \bot\} \\
 & [\![r_1 \mid\mid r_2]\!]_\bot & = & [\![r_1]\!]_\bot \,\&\, [\![r_2]\!]_\bot \\
\boxed{[\![R]\!]} & [\![\diamond]\!] & = & \top \\
 & [\![r, R]\!] & = & [\![r]\!] \,\&\, [\![r]\!]_\bot \,\&\, [\![R]\!] \\
\boxed{[\![T]\!]} & [\![\diamond]\!] & = & \bullet \\
 & [\![T, \alpha \,\#\, R]\!] & = & [\![T]\!], \alpha * [\![R]\!], \alpha_\bot * [\![R]\!]
\end{array}
$$

Figure 6 Elaboration functions from $\lambda^{||}$ to F_i^+.

regularly elaborated types as well as the bottom-elaborated types into the disjointness constraint. In other words, β is disjoint with both α and $\alpha_\bot$. Now ε_2 elaborates to E_2', which type-checks successfully in F_i^+. Note we have also elaborated and bottom-elaborated Empty.

$$E_2' = \Lambda(\alpha * \top \,\&\, \top).\, \Lambda(\alpha_\bot * \top \,\&\, \top).\, \Lambda(\beta * \alpha \,\&\, \alpha_\bot).\, \Lambda(\beta_\bot * \alpha \,\&\, \alpha_\bot).\, \lambda x : \alpha.\, \lambda y : \beta.\, x \,,,\, y$$

4.5 Formal Elaboration

With all the above ideas and observations in mind, we are ready to give a formal account of the elaboration. The elaboration of types is given in Figure 6. We highlight the differences from Figure 5 in grey. There are two ways of elaborating records: $[\![r]\!]$ (contained in $[\![t]\!]$) for regular elaboration and $[\![r]\!]_\bot$ for bottom elaboration. In regular elaboration $[\![t]\!]$, α elaborates to α. Of particular interest is the case of elaborating quantifiers: each quantifier $\forall \alpha \,\#\, R.\, t$ is split into two quantifiers $\forall(\alpha * [\![R]\!]).\, \forall(\alpha_\bot * [\![R]\!]).\, [\![t]\!]$ in F_i^+. The relative order of α and $\alpha_\bot$ is not important, as long as we respect the order when elaborating type applications. Bottom elaboration $[\![r]\!]_\bot$ elaborates α to $\alpha_\bot$, and $\{l : t\}$ to $\{l : \bot\}$.

When elaborating constraint lists ($[\![R]\!]$), a record r is elaborated to the intersection of both its regular elaboration and bottom elaboration. Thus if β is compatible with α, then its elaboration β is disjoint with both α and $\alpha_\bot$.

Now let us go back to the gray parts in Figure 4. The major difference from Figure 5 is rule WTT-ALLI and rule WTT-ALLE. In rule WTT-ALLI, we elaborate constrained type abstractions to disjoint type abstractions with two quantifiers, matching the elaboration of constrained quantification. Note that the relative order of α and $\alpha_\bot$ should match the order of α and $\alpha_\bot$ in elaborating quantifiers. Similarly, in the type application $\varepsilon\ [r]$ (rule WTT-ALLE), we first elaborate e to E. The elaboration E is then applied to two types $[\![r]\!]$ and $[\![r]\!]_\bot$, as E has two quantifiers resulting from the elaboration. It is of great importance that the relative order of $[\![r]\!]$ and $[\![r]\!]_\bot$ should match the order of α and $\alpha_\bot$ in elaborating quantifiers. There is a *protocol* that we must follow during elaboration: if α is substituted by $[\![r]\!]$, then $\alpha_\bot$ is substituted by $[\![r]\!]_\bot$.

4.6 Metatheory

Our elaboration enjoys desired properties. The following lemma states that our elaboration function commutes with substitution, in a slightly involved way:

▶ **Lemma 10** (Elaboration commutes with substitution). *We have* **1.** $[\![[r/\alpha]t]\!] = [[\![r]\!]_\perp/\alpha_\perp][[\![r]\!]/\alpha][\![t]\!]$; **2.** $[\![[r/\alpha]r_1]\!]_\perp = [[\![r]\!]_\perp/\alpha_\perp][[\![r]\!]/\alpha][\![r_1]\!]_\perp$; *and* **3.** $[\![[r/\alpha]R]\!] = [[\![r]\!]_\perp/\alpha_\perp][[\![r]\!]/\alpha][\![R]\!]$.

We show key lemmas that *bridge the gap* between row and disjoint polymorphism.

▶ **Lemma 11** (Type equivalence implies subtyping). *If* $t_1 \sim t_2$, *then we have* $[\![t_1]\!] <: [\![t_2]\!]$ *and* $[\![t_2]\!] <: [\![t_1]\!]$.

▶ **Lemma 12** (Compatibility implies disjointness). *If* $T \vdash r_1 \# r_2$, *then we have:* **1.** $[\![T]\!] \vdash [\![r_1]\!] * [\![r_2]\!]$; **2.** $[\![T]\!] \vdash [\![r_1]\!] * [\![r_2]\!]_\perp$; **3.** $[\![T]\!] \vdash [\![r_1]\!]_\perp * [\![r_2]\!]$; *and* **4.** $[\![T]\!] \vdash [\![r_1]\!]_\perp * [\![r_2]\!]_\perp$.

▶ **Lemma 13** (Essence of compatibility). *If* $T \vdash r \# \{l : t\}$, *then for all* A, *we have* **1.** $[\![T]\!] \vdash [\![r]\!] * \{l : A\}$; *and* **2.** $[\![T]\!] \vdash [\![r]\!]_\perp * \{l : A\}$.

With everything in place, we prove that our elaboration in Figure 4 is type-safe. The reader can refer to our Coq formalization for details.

▶ **Theorem 14** (Type-safety of elaboration). *If* $T; G \vdash \varepsilon : t \rightsquigarrow E$, *then* $[\![T]\!]; [\![G]\!] \vdash E \Rightarrow [\![t]\!]$.

Coherence Because of rule WTT-EQ, a $\lambda^{||}$ expression can possibly elaborate to many different F_i^+ expressions. For example, the term $\Lambda(\alpha \# \{l : \mathsf{Int}\}). \lambda(x : \alpha). x$ has the following two elaborations E_1 and E_2 (among others). This is the problem of coherence [54]: the meaning of a target program depends on the choice of a particular elaboration typing.

1. $E_1 = \Lambda(\alpha * (\{l : \mathsf{Int}\} \,\&\, \{l : \perp\})). \Lambda(\alpha_\perp * (\{l : \mathsf{Int}\} \,\&\, \{l : \perp\})). \lambda(x : \alpha). x$;
2. $E_2 = (E_1 : [\![\forall\alpha \# \{l : \mathsf{Bool}\}. \alpha \to \alpha]\!]) : [\![\forall\alpha \# \{l : \mathsf{Int}\}. \alpha \to \alpha]\!]$

To prove that different elaborations are *equivalent*, we utilize the definition of *contextual equivalence*. In particular, we prove that if a $\lambda^{||}$ expression ε with type t elaborates to two F_i^+ expressions, and these two F_i^+ expressions further elaborate to two F_{co} expressions, then the F_{co} expressions are contextually equivalent.

▶ **Theorem 15** (Coherence of elaboration). *If* $\diamond; \diamond \vdash \varepsilon : t \rightsquigarrow E_1$, *and* $\diamond; \diamond \vdash \varepsilon : t \rightsquigarrow E_2$, *and* $\bullet; \bullet \vdash E_1 \Rightarrow [\![t]\!] \rightsquigarrow e_1$, *and* $\bullet; \bullet \vdash E_2 \Rightarrow [\![t]\!] \rightsquigarrow e_2$, *then* $\bullet; \bullet \vdash e_1 \backsimeq_{ctx} e_2$.

5 Encoding Bounded Quantification

This section presents a type-safe and coherent encoding of kernel $\mathsf{F}_{<:}$ [12] into F_i^+. This encoding validates the informal observation about the relationship between polymorphic intersection systems and bounded quantification.

5.1 Syntax and Semantics of kernel $\mathsf{F}_{<:}$

We start by reviewing the syntax and semantics of kernel $\mathsf{F}_{<:}$, a polymorphic calculus with bounded quantification. The syntax of $\mathsf{F}_{<:}$ is given at the top of Figure 7. It is a version of $\mathsf{F}_{<:}$ extended with records[3] [10]. In addition to standard System F constructs, types σ include bounded quantifications $\forall(\alpha <: \tau). \sigma$, which give a *bound* for the type variable; and record types $\{l_1 : \sigma_1, .., l_n : \sigma_n\}$, for which we assume all labels are distinct. In addition to standard System F terms, terms ϵ include type abstractions $\Lambda(\alpha <: \sigma). \epsilon$, records $\{l_1 = \epsilon_1, .., l_n = \epsilon_n\}$, and projections $\epsilon.l$. Contexts Σ record both the types of term variables, and the bounds of type variables. We use $\Sigma \vdash \sigma$ to mean that a type is well-formed under a context.

[3] We could also encode record types in $\mathsf{F}_{<:}$, which however is a bit involved.

Types	σ, τ	$::=$	$\mathsf{Int} \mid \top \mid \alpha \mid \sigma \to \tau \mid \forall (\alpha <: \tau).\, \sigma \mid \{l_1 : \sigma_1, \,..\, , l_n : \sigma_n\}$
Terms	ϵ	$::=$	$i \mid \top \mid x \mid \lambda(x : \sigma).\, \epsilon \mid \epsilon_1\, \epsilon_2 \mid \Lambda(\alpha <: \tau).\, \epsilon \mid \epsilon\, \sigma \mid \{l_1 = \epsilon_1, \,..\, , l_n = \epsilon_n\} \mid \epsilon.l$
Value	υ	$::=$	$i \mid \top \mid \lambda(x : \sigma).\, \epsilon \mid \Lambda(\alpha <: \sigma).\, \epsilon \mid \{l_1 = \upsilon_1, \,..\, , l_n = \upsilon_n\}$
Context	Σ	$::=$	$\diamond \mid \Sigma, x : \sigma \mid \Sigma, \alpha <: \sigma$

$\boxed{\Sigma \vdash \sigma <: \tau}$ *(Subtyping)*

$$\textsc{F-sub-refl}\quad \frac{\Sigma\ \mathsf{ok} \qquad \Sigma \vdash \sigma}{\Sigma \vdash \sigma <: \sigma} \qquad \textsc{F-sub-trans}\quad \frac{\Sigma \vdash \sigma_1 <: \sigma_2 \qquad \Sigma \vdash \sigma_2 <: \sigma_3}{\Sigma \vdash \sigma_1 <: \sigma_3}$$

$$\textsc{F-sub-top}\quad \frac{\Sigma\ \mathsf{ok} \qquad \Sigma \vdash \sigma}{\Sigma \vdash \sigma <: \top} \qquad \textsc{F-sub-tvar-binds}\quad \frac{(\alpha <: \sigma) \in \Sigma}{\Sigma \vdash \alpha <: \sigma}$$

$$\textsc{F-sub-arrow}\quad \frac{\Sigma \vdash \tau_1 <: \sigma_1 \qquad \Sigma \vdash \sigma_2 <: \tau_2}{\Sigma \vdash \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2} \qquad \textsc{F-sub-forall}\quad \frac{\Sigma, \alpha <: \tau \vdash \sigma_1 <: \sigma_2}{\Sigma \vdash \forall (\alpha <: \tau).\, \sigma_1 <: \forall (\alpha <: \tau).\, \sigma_2}$$

$$\textsc{F-sub-rcdDepth}\quad \frac{\text{for each } i \qquad \Sigma \vdash \sigma_i <: \tau_i}{\Sigma \vdash \{l_i : \sigma_i^{\,i \in 1..n}\} <: \{l_i : \tau_i^{\,i \in 1..n}\}}$$

$$\textsc{F-sub-rcdWidth}\quad \frac{}{\Sigma \vdash \{l_i : \sigma_i^{\,i \in 1..n+k}\} <: \{l_i : \sigma_i^{\,i \in 1..n}\}} \qquad \textsc{F-sub-rcdPerm}\quad \frac{\{l'_j : \tau_j^{\,j \in 1..n}\} \text{ is a permutation of } \{l_i : \sigma_i^{\,i \in 1..n}\}}{\Sigma \vdash \{l'_j : \tau_j^{\,j \in 1..n}\} <: \{l_i : \sigma_i^{\,i \in 1..n}\}}$$

$\boxed{\Sigma \vdash \epsilon : \sigma \rightsquigarrow E}$ *(Typing)*

$$\textsc{F-top}\quad \frac{\Sigma\ \mathsf{ok}}{\Sigma \vdash \top : \top \rightsquigarrow \top} \qquad \textsc{F-nat}\quad \frac{\Sigma\ \mathsf{ok}}{\Sigma \vdash i : \mathsf{Int} \rightsquigarrow i} \qquad \textsc{F-var}\quad \frac{\Sigma\ \mathsf{ok} \qquad (x : \sigma) \in \Sigma}{\Sigma \vdash x : \sigma \rightsquigarrow x}$$

$$\textsc{F-arrow}\quad \frac{\Sigma, x : \sigma \vdash \epsilon : \tau \rightsquigarrow E}{\Sigma \vdash \lambda(x : \sigma).\, \epsilon : \sigma \to \tau \rightsquigarrow (\lambda x.\, E) : (\llbracket \sigma \rrbracket_\Sigma \to \llbracket \tau \rrbracket_\Sigma)} \qquad \textsc{F-sub}\quad \frac{\Sigma \vdash \epsilon : \sigma \rightsquigarrow E \qquad \Sigma \vdash \sigma <: \tau}{\Sigma \vdash \epsilon : \tau \rightsquigarrow E : \llbracket \tau \rrbracket_\Sigma}$$

$$\textsc{F-app}\quad \frac{\Sigma \vdash \epsilon_1 : \sigma \to \tau \rightsquigarrow E_1 \qquad \Sigma \vdash \epsilon_2 : \sigma \rightsquigarrow E_2}{\Sigma \vdash \epsilon_1\, \epsilon_2 : \tau \rightsquigarrow E_1\, E_2} \qquad \textsc{F-tabs}\quad \frac{\Sigma, \alpha <: \sigma \vdash \epsilon : \tau \rightsquigarrow E}{\Sigma \vdash \Lambda(\alpha <: \sigma).\, \epsilon : \forall(\alpha <: \sigma).\, \tau \rightsquigarrow \Lambda(\alpha * \top).\, E}$$

$$\textsc{F-rcd}\quad \frac{\Sigma \vdash \epsilon_1 : \sigma_1 \rightsquigarrow E_1 \,..\, \Sigma \vdash \epsilon_n : \sigma_n \rightsquigarrow E_n}{\Sigma \vdash \{l_1 = \epsilon_1, \,..\, , l_n = \epsilon_n\} : \{l_1 : \sigma_1, \,..\, , l_n : \sigma_n\} \rightsquigarrow \{l_1 = E_1\}, , \,..\, , \{l_n = E_n\}}$$

$$\textsc{F-proj}\quad \frac{\Sigma \vdash \epsilon : \{l_1 : \sigma_1, \,..\, , l : \sigma, \,..\, , l_n : \sigma_n\} \rightsquigarrow E}{\Sigma \vdash \epsilon.l : \sigma \rightsquigarrow (E : \llbracket \{l : \sigma\} \rrbracket_\Sigma).l} \qquad \textsc{F-tapp}\quad \frac{\Sigma \vdash \epsilon : \forall(\alpha <: \tau_1).\, \tau_2 \rightsquigarrow E \qquad \Sigma \vdash \sigma <: \tau_1}{\Sigma \vdash \epsilon\, \sigma : [\sigma/\alpha]\tau_2 \rightsquigarrow (E\, \llbracket \sigma \rrbracket_\Sigma) : (\llbracket ([\sigma/\alpha]\tau_2) \rrbracket_\Sigma)}$$

$$\boxed{\llbracket \sigma \rrbracket_\Sigma} \qquad \begin{aligned} \llbracket \mathsf{Int} \rrbracket_\Sigma &= \mathsf{Int} \\ \llbracket \top \rrbracket_\Sigma &= \top \\ \llbracket (\sigma \to \tau) \rrbracket_\Sigma &= \llbracket \sigma \rrbracket_\Sigma \to \llbracket \tau \rrbracket_\Sigma \\ \llbracket (\{l_1 : \sigma_1, \,..\, , l_n : \sigma_n\}) \rrbracket_\Sigma &= \{l_1 : \llbracket \sigma_1 \rrbracket_\Sigma\} \,\&\, \,..\, \,\&\, \{l_n : \llbracket \sigma_n \rrbracket_\Sigma\} \\ \llbracket \alpha \rrbracket_{(\Sigma, x : \sigma)} &= \llbracket \alpha \rrbracket_\Sigma \\ \llbracket \alpha \rrbracket_{(\Sigma, \beta <: \sigma)} &= \llbracket \alpha \rrbracket_\Sigma \\ \llbracket \alpha \rrbracket_{(\Sigma, \alpha <: \sigma)} &= \alpha \,\&\, \llbracket \sigma \rrbracket_\Sigma \\ \llbracket \forall(\alpha <: \sigma).\, \tau \rrbracket_\Sigma &= \forall(\alpha * \top).\, \llbracket \tau \rrbracket_{\Sigma, \alpha <: \sigma} \end{aligned}$$

$$\boxed{\llfloor \Sigma \rrfloor} \qquad \begin{aligned} \llfloor \diamond \rrfloor &= \bullet \\ \llfloor \Sigma, \alpha <: \sigma \rrfloor &= \llfloor \Sigma \rrfloor, \alpha * \top \\ \llfloor \Sigma, x : \sigma \rrfloor &= \llfloor \Sigma \rrfloor \end{aligned} \qquad \boxed{\llceil \Sigma \rrceil} \qquad \begin{aligned} \llceil \diamond \rrceil &= \bullet \\ \llceil \Sigma, \alpha <: \sigma \rrceil &= \llceil \Sigma \rrceil \\ \llceil \Sigma, x : \sigma \rrceil &= \llceil \Sigma \rrceil, x : \llbracket \sigma \rrbracket_\Sigma \end{aligned}$$

Figure 7 Syntax, subtyping, typing and elaboration of kernel $\mathsf{F}_{<:}$.

Subtyping The subtyping relation is presented in the middle of Figure 7. Most rules are quite standard. Rule F-SUB-TVAR-BINDS says that a type variable α is a subtype of its bound σ. Rule F-SUB-FORALL, first introduced in Fun [12], requires that the bounds of two quantified types must be identical in order for one to be a subtype of the other. Full $\mathsf{F}_{<:}$ relaxes this restriction and includes a more powerful formulation where subtyping of quantified types is contravariant in their bounds and covariant in their bodies. We will discuss full $\mathsf{F}_{<:}$ in Section 6.2. Rules F-SUB-RCDDEPTH, F-SUB-RCDWIDTH, and F-SUB-RCDPERM together form the usual record subtyping.

Typing The typing rules of $\mathsf{F}_{<:}$ are shown below the subtyping relation. The reader is advised to ignore the gray parts for now. Most rules are straightforward. Unlike F_i^+, $\mathsf{F}_{<:}$ has a subsumption rule (rule F-SUB) for implicit upcasting that can be triggered anywhere during type-checking. Type abstractions are checked by moving their bounds into the context (rule F-TABS), and type applications check that the type being passed satisfies the bound of the corresponding quantifier (rule F-TAPP).

5.2 Elaboration Function

Adapting the encoding from Pierce [47] to our setting, we have

$$\forall(\alpha <: \sigma).\, \tau \triangleq \forall(\alpha * \top).\ [\alpha \,\&\, \sigma/\alpha]\tau$$

We turn the encoding into an elaboration function. Instead of immediately substituting α with $\alpha \,\&\, \sigma$, we collect the bounds $\alpha <: \sigma$ as we traverse the quantifiers, and only substitute when we encounter a type variable α. This strategy is consistent with elaborating types with free type variables. For example, consider the expression $\alpha <: \mathsf{Int} \vdash (\lambda(x : \alpha).\, x+1) : \alpha \to \mathsf{Int}$. This expression type-checks because we have the information $\alpha <: \mathsf{Int}$ in the context so that we can upcast (by rule F-SUB) the type of x to Int when checking $x + 1$. What is of importance here is to propagate the context information to the type being elaborated. In a fairly standard way, we regard the context as a big binder. Intuitively, if we elaborate α under the context $\alpha <: \mathsf{Int}$, it should give us the same result as if elaborating α in the body of $\forall(\alpha <: \mathsf{Int}).\, \alpha$. Therefore, in this case, we substitute α by $\alpha \,\&\, \mathsf{Int}$, which yields $x : \alpha \,\&\, \mathsf{Int}$, and thus validates $x + 1$.

Formally, type elaboration is denoted as $[\![\sigma]\!]_\Sigma = A$, which reads: under context Σ, type σ elaborates to type A. Elaboration of a closed type is just a special case where the context is empty, i.e., $[\![\sigma]\!]_\diamond$. The full definition is given on the lower left of Figure 7. Most rules are self-explanatory. In particular, bounded quantification elaborates into disjoint quantification by moving the bound information into the context. When elaborating a type variable α, we traverse the context until we find its subtyping constraint $\alpha <: \sigma$, and then we substitute it with an intersection type $\alpha \,\&\, [\![\sigma]\!]_\Sigma$.

▶ **Lemma 16** ($[\![\sigma]\!]_\Sigma$ is total). *If $\Sigma \vdash \sigma$, then there exists a unique type A such that $[\![\sigma]\!]_\Sigma = A$.*

We now lift the elaboration function to contexts, given on the lower right of Figure 7. $\lfloor\!\lfloor \Sigma \rfloor\!\rfloor$ elaborates a $\mathsf{F}_{<:}$ context to a F_i^+ type context, in which subtyping constraints $\alpha <: \sigma$ of type variables are elaborated to disjointness constraints $\alpha * \top$ and all term variables are ignored. $\lceil\!\lceil \Sigma \rceil\!\rceil$ elaborates a $\mathsf{F}_{<:}$ context to a F_i^+ term context, in which all type variables are ignored and the types of term variables are elaborated under the prefix context.

5.3 Type-directed Elaboration

An intuitive elaboration scheme of expressions is to simply apply the elaboration function to types. For example, under context Σ, if ϵ elaborates to E, then type applications $\epsilon\,\sigma$ elaborates to $E\,[\![\sigma]\!]_\Sigma$. Now let us consider an example.

▶ **Example 17.** Consider a $\mathsf{F}_{<:}$ judgment

$$\beta <: \mathsf{Int} \vdash (\Lambda(\alpha <: \top).\, \lambda(x : \alpha).\, x)\, \beta : \beta \to \beta$$

Here the type application type-checks because by rule F-SUB-TOP we have $\beta <: \top$. If we elaborate $\epsilon\,\sigma$ to $E\,[\![\sigma]\!]_\Sigma$ directly, the resulting expression is

$$(\Lambda(\alpha * \top).\,(\lambda x.\, x) : (\alpha \,\&\, \top) \to (\alpha \,\&\, \top))\,(\beta \,\&\, \mathsf{Int})$$

Note that as F_i^+ does not have annotated abstractions, we put the elaborated arrow type as the type annotation. Following the typing rule of F_i^+, we can infer the type of this expression:

$$\beta * \top; \bullet \vdash (\Lambda(\alpha * \top).\,((\lambda x.\, x) : (\alpha \,\&\, \top) \to (\alpha \,\&\, \top))\,(\beta \,\&\, \mathsf{Int})) \Rightarrow (\beta \,\&\, \mathsf{Int} \,\&\, \top) \to (\beta \,\&\, \mathsf{Int} \,\&\, \top)$$

However, the expected result type $\beta \to \beta$ elaborates to

$$(\beta \,\&\, \mathsf{Int}) \to (\beta \,\&\, \mathsf{Int})$$

Now we get a mismatch between the actual type $((\beta \,\&\, \mathsf{Int} \,\&\, \top) \to (\beta \,\&\, \mathsf{Int} \,\&\, \top))$ and the expected type $((\beta \,\&\, \mathsf{Int}) \to (\beta \,\&\, \mathsf{Int}))$ of the expression!

Fortunately, in this particular example, we can prove that the actual type and the expected type are subtypes of each other, i.e., they are isomorphic. Why is that true? Recall that we have $\beta <: \top$, which after elaboration gives us $(\beta \,\&\, \mathsf{Int}) <: \top$. Therefore we can show that the following two subtyping instances are valid: **1.** $(\beta \,\&\, \mathsf{Int} \,\&\, \top) \to (\beta \,\&\, \mathsf{Int} \,\&\, \top) <: (\beta \,\&\, \mathsf{Int}) \to (\beta \,\&\, \mathsf{Int})$; and **2.** $(\beta \,\&\, \mathsf{Int}) \to (\beta \,\&\, \mathsf{Int}) <: (\beta \,\&\, \mathsf{Int} \,\&\, \top) \to (\beta \,\&\, \mathsf{Int} \,\&\, \top)$

More generally, we prove that elaboration commutes with substitution, yielding isomorphic types. Consider that under the context Σ, we have a type application $\epsilon\,\sigma$, where ϵ has type $\forall(\alpha <: \tau_1).\,\tau_2$, and in order for it to type-check, we have $\sigma <: \tau_1$. The expected type we want of the expression is the elaboration of the $\mathsf{F}_{<:}$ typing result, i.e., $[\![([\sigma/\alpha]\tau_2)]\!]_\Sigma$. The actual type is the result of feeding the elaborated argument $[\![\sigma]\!]_\Sigma$ to the elaborated quantification $[\![\forall(\alpha <: \tau_1).\,\tau_2]\!]_\Sigma$, i.e., $[[\![\sigma]\!]_\Sigma/\alpha]([\![\tau_2]\!]_{(\Sigma,\alpha<:\tau_1)})$.

▶ **Lemma 18** (Elaboration commutes with substitution)**.** *Given* $\Sigma \vdash \sigma <: \tau_1$*, we have* **1.** $[\![[\sigma/\alpha]\tau_2]\!]_\Sigma <: [[\![\sigma]\!]_\Sigma/\alpha]([\![\tau_2]\!]_{(\Sigma,\alpha<:\tau_1)})$*; and* **2.** $[[\![\sigma]\!]_\Sigma/\alpha]([\![\tau_2]\!]_{(\Sigma,\alpha<:\tau_1)}) <: [\![([\sigma/\alpha]\tau_2)]\!]_\Sigma$.

Note that *the elaboration scheme slightly varies depending on the type semantics of the target intersection type calculi.* It is a desirable property that typing should be preserved after elaboration, i.e., the elaborated expression should have the corresponding elaborated type. For languages with an implicit subsumption rule (e.g., rule F-SUB in kernel $\mathsf{F}_{<:}$), Lemma 18 can implicitly upcast the actual type to the expected type, and thus validates the intuitive elaboration of the type applications. For languages with *explicit* subsumption rules (e.g., rule T-SUB in F_i^+), to remedy this situation, we need to *annotate the expression with the expected type* to explicitly upcast the type. Concretely, in this example, the elaborated expression, with the added annotation highlighted in grey, will be:

$$((\Lambda(\alpha * \top).\,(\lambda x.\, x) : (\alpha \,\&\, \top) \to (\alpha \,\&\, \top))\,(\beta \,\&\, \mathsf{Int})) : (\beta \,\&\, \mathsf{Int}) \to (\beta \,\&\, \mathsf{Int})$$

Finally, we can go back and consider the elaboration of expressions in the grey part of Figure 7. Most of the elaboration rules are self-explanatory. In particular, in rule F-TAPP, type applications $\epsilon\,\sigma$ elaborates to $(E\,[\![\sigma]\!]_\Sigma) : [\![([\sigma/\alpha]\tau_2)]\!]_\Sigma$.

5.4 Metatheory

Now that we have everything in place, we are ready to prove that our elaboration is sound.

▶ **Theorem 19** (Type-safety of elaboration)**.** *If* $\Sigma \vdash \epsilon : \sigma \rightsquigarrow E$*, then* $\lfloor\!\lfloor \Sigma \rfloor\!\rfloor; \lceil\!\lceil \Sigma \rceil\!\rceil \vdash E \Rightarrow [\![\sigma]\!]_\Sigma$.

$$\begin{tikzcd}
\text{kernel } \mathsf{F}_{<:} & (\Lambda(\alpha <: \mathsf{Int}).\,\lambda(x : \alpha).\,1)\,\mathsf{Int} \arrow[rrr] \arrow[d, squiggly] & & & \lambda(x : \mathsf{Int}).\,1 \arrow[d, squiggly] \\
\mathsf{F}_i^+ & ((\Lambda(\alpha * \top).\,((\lambda x.\,1) : \alpha \,\&\, \mathsf{Int} \to \mathsf{Int}))\,\mathsf{Int}) : \mathsf{Int} \to \mathsf{Int} \arrow[d, squiggly] & & & (\lambda x.\,1) : \mathsf{Int} \to \mathsf{Int} \arrow[d, squiggly] \\
\mathsf{F}_{co} & ((\langle \mathsf{id}, \mathsf{id} \rangle \to \mathsf{id})\,((\Lambda\alpha.\,\lambda x.\,1)\,\mathsf{Int}) \arrow[r] & ((\langle \mathsf{id}, \mathsf{id} \rangle \to \mathsf{id})\,(\lambda x.\,1) \arrow[rr, phantom, "\backsimeq_{ctx}"] & & \lambda x.\,1
\end{tikzcd}$$

Figure 8 Key idea of simulation illustrated with an example.

However, due to the implicit upcasting (rule F-SUB), a $\mathsf{F}_{<:}$ expression can possibly elaborate to many different ones in F_i^+. For example, consider $(\lambda(x : \top).\,2)\,1$. Two elaborations (among others) are **1.** $((\lambda x.\,2) : \top \to \mathsf{Int})\,(1 : \top)$; and **2.** $(((\lambda x.\,2) : \top \to \mathsf{Int}) : \mathsf{Int} \to \mathsf{Int})\,1$. Therefore, we prove that different elaborations lead to *contextually equivalent* results.[4]

▶ **Theorem 20** (Coherence of elaboration). *If* $\diamond \vdash \epsilon : \sigma \rightsquigarrow E_1$ *, and* $\diamond \vdash \epsilon : \sigma \rightsquigarrow E_2$ *, and* $\bullet;\bullet \vdash E_1 \Rightarrow [\![\sigma]\!]_\diamond \rightsquigarrow e_1$ *, and* $\bullet;\bullet \vdash E_2 \Rightarrow [\![\sigma]\!]_\diamond \rightsquigarrow e_2$ *, then* $\bullet;\bullet \vdash e_1 \backsimeq_{ctx} e_2$.

We also prove a weaker *simulation* result[5]: if the standard direct operational semantics of kernel $\mathsf{F}_{<:}$ produces $\epsilon_1 \longrightarrow \epsilon_2$, and ϵ_2 elaborates to E_2 in F_i^+, which in turn elaborates to e_2 in F_{co}, then ϵ_1 elaborates to E_1 in F_i^+, which in turn elaborates to e_1 in F_{co}, and $e_1 \longrightarrow e_1'$, *where* e_1' *and* e_2 *are contextually equivalent.* The lemma is weaker in the sense that e_1' and e_2 are not syntactically equivalent. Given the coherence lemmas of F_i^+ and of the elaboration, it is no surprise that here contextual equivalence takes the place of the syntactic equivalence, as explicit upcasting generates coercions, which may break syntactic equivalence. As an example, consider Figure 8, where e_1 steps to an expression $e_1' = ((\langle \mathsf{id}, \mathsf{id} \rangle \to \mathsf{id})\,(\lambda x.\,1)$ that is contextually equivalent to $e_2 = \lambda x.\,1$.

▶ **Theorem 21** (Simulation). *If* $\epsilon_1 \longrightarrow \epsilon_2$*, and* $\diamond \vdash \epsilon_2 : \sigma \rightsquigarrow E_2$ *, and* $\bullet;\bullet \vdash E_2 \Rightarrow [\![\sigma]\!]_\diamond \rightsquigarrow e_2$ *, then there exist* E_1*,* e_1*,* e_1' *such that* $\diamond \vdash \epsilon_1 : \sigma \rightsquigarrow E_1$ *, and* $\bullet;\bullet \vdash E_1 \Rightarrow [\![\sigma]\!]_\diamond \rightsquigarrow e_1$ *, and* $e_1 \longrightarrow e_1'$*, where* $\bullet;\bullet \vdash e_1' \backsimeq_{ctx} e_2$.

The detailed paper proof of this lemma is given in Appendix D. This lemma requires a generalized logical equivalence for F_i^+, which is not yet supported in the current Coq framework. Therefore we only present the paper proof. If the Coq framework of F_i^+ is generalized, we expect that the lemma can be proved in Coq.

6 Discussion

In this section we discuss some possible paths for further exploration.

6.1 Variants of Row Polymorphism

According to Rémy [52], record calculi can typically be categorized into two groups based on how they support the extension operation: the *strict* group does not allow duplicate labels,

[4] One restriction in Bi et al. [7] is that due to the well-foundedness issue, the logical relation of F_i^+ is defined only for its *predicative* subset, where type arguments in type applications can only be monotypes. Since our proof is built upon the logical relation of F_i^+, Theorem 20 is restricted to predicative subset of kernel $\mathsf{F}_{<:}$ as well. If the well-foundedness of impredicative F_i^+ is recovered, e.g., by employing step-indexing logical relations [1], we expect that our proof remains valid.

[5] Note that $\lambda^{||}$ does not provide a semantics [34], so we did not discuss the operational semantics in Section 4. If $\lambda^{||}$ had a operational semantics, we believe a similar theorem would apply.

while the *free* group does. We have already shown that F_i^+ supports $\lambda^{||}$, a calculus in the strict group, with a more fine-grained control as disjointness allows duplicate labels as long as their types are disjoint. λ^{TIR} [57] is another calculus from the strict group, which introduces type-indexed rather than label-indexed rows, and uses *membership constraints* to avoid conflicts. To distinguish types and row, λ^{TIR} incorporates a kind system that distinguishes rows from types. We believe that F_i^+ could also serve as a target for λ^{TIR}, as type-indexed rows are closely related to disjoint intersections. Thus an elaboration from λ^{TIR} to F_i^+ is interesting future work.

For the free group, there are two different approaches for extension: previous fields are always retained, and record projections always select the first matching label [35]; or the extension overwrites the field if it is already present [5, 52, 11]. The former system suffers from the similar issue of *ambiguity*, as records can be extended with the same label even when types are overlapping, which violates the essence of disjointness. For the latter system, essentially F_i^+ is capable to encode the extension operation in a different form. Consider a function that overwrites ($\leftarrow$) the label l in a record by incrementing the original value [11]:

$$inc = \Lambda\alpha <: \{l : \mathsf{Int}\}.\lambda(x : \alpha).\, x \leftarrow \{l = x.l + 1\}$$

In F_i^+, we can define

$$inc' = \Lambda(\alpha * \{l : \mathsf{Int}\}).\, \lambda(x : \alpha \,\&\, \{l : \mathsf{Int}\}).\, (x : \alpha \,,,\, \{l = (x : \{l : \mathsf{Int}\}).l + 1\})$$

There are two differences. Firstly, the type arguments to the two functions are different: *inc* expects a type argument which includes $\{l : \mathsf{Int}\}$, while inc' expects a type argument which excludes $\{l : \mathsf{Int}\}$, and $\{l : \mathsf{Int}\}$ is later recovered in x's type by an intersection type. This explains a more involved encoding. Secondly, the term arguments to the two functions are also different: *inc* accepts arguments that have exactly one l label with type Int, while inc' can accept arguments of type $\{l : \mathsf{Int}\} \,\&\, \{l : \mathsf{Bool}\}$. This again manifests the fine-grained control of disjointness. That being said, we have not studied nor formalized the encoding.

Type-inference The focus of our work is languages that have more modest goals in terms of type-inference. Note that neither $\lambda^{||}$ or F_i^+ address sophisticated type-inference. We focus on languages with subtyping, including TypeScript, Ceylon, Scala or Flow. Languages like Racket also include a variant of row polymorphism, without full-type inference to model powerful OOP features [58]. Many other row type systems [52, 61, 60, 35] support type inference. For the future, we wish to investigate whether a disjoint polymorphic calculus offering similar type inference can model calculi with row polymorphism and type inference. We believe that several ideas employed in work on type inference for row polymorphism can be adapted to a setting with disjoint polymorphism.

6.2 Variants of Bounded Quantification

Full $\mathsf{F}_{<:}$ [23] includes a more powerful formulation of subtyping for universal quantification (rule F-SUB-FORALLALT), which is contravariant in the bound types and covariant in the body types. However, this subtyping rule renders subtyping in full $\mathsf{F}_{<:}$ undecidable [48].

$$\frac{\Sigma \vdash \tau_2 <: \tau_1 \qquad \Sigma, \alpha <: \tau_2 \vdash \sigma_1 <: \sigma_2}{\Sigma \vdash \forall(\alpha <: \tau_1).\, \sigma_1 <: \forall(\alpha <: \tau_2).\, \sigma_2}\ \textsc{f-sub-forallAlt}$$

Moreover, this rule breaks the encoding. Consider the example [47]:

$$\diamond \vdash \forall(\alpha <: \top).\, \alpha <: \forall(\alpha <: \mathsf{Int}).\, \alpha$$

which elaborates to a non-derivable F_i^+ judgment

$\bullet \vdash \forall(\alpha * \top).\, \alpha \,\&\, \top <: \forall(\alpha * \top).\, \alpha \,\&\, \mathsf{Int}$

since $\alpha * \top \vdash \alpha \,\&\, \top <: \alpha \,\&\, \mathsf{Int}$ is not true.

One possible solution is to adopt a more powerful subtyping relation in the target calculus, where a polymorphic type is a subtype of another one if the first has more instances [45]. For example, we can have the following judgment, as α can be instantiated to Int to get the right hand side:

$\forall\alpha.\, \alpha \to \alpha <: \mathsf{Int} \to \mathsf{Int}$

Then the judgment $\bullet \vdash \forall(\alpha * \top).\, \alpha \,\&\, \top <: \forall(\alpha * \top).\, \alpha \,\&\, \mathsf{Int}$ is derivable. After we skolemise the type variable α in the right hand side, we can instantiate α in the left hand side by $\alpha \,\&\, \mathsf{Int}$ to get $\alpha * \top \vdash \alpha \,\&\, \mathsf{Int} \,\&\, \top <: \alpha \,\&\, \mathsf{Int}$.

Interestingly, such subtyping is usually *predicative*, i.e., universal quantifications can only be instantiated with monotypes; or otherwise it is undecidable. Thus if the bounds can only be monotypes, it may be the case that a target calculus with the more powerful subtyping rule can encode the predicative version of full $\mathsf{F}_{<:}$.

6.3 Variants of Intersection Type Systems

$\lambda^{||}$ is encodable into intersection type systems that feature the merge operator, unrestricted intersection types, polymorphism and guarantee coherence through constraints similar to compatibility or disjointness. This currently only applies to F_i^+. Some intersection type systems [28, 6, 46] only support simple record types. While Alpuim and Oliveira [2] do support polymorphism, they only allow intersection types between disjoint types. Hence, our elaboration of constraint lists to $[\![r]\!] \,\&\, [\![r]\!]_\perp$ is rejected as $[\![r]\!]$ and $[\![r]\!]_\perp$ may not be disjoint.

Kernel $\mathsf{F}_{<:}$ is encodable for intersection type systems that feature polymorphism and unrestricted intersection types. For example, a similar encoding might be applicable to other intersection type systems [17, 19]. Interestingly, the behavior of elaborated expressions varies according to the type semantics of the target. Consider a function f of type $\forall(\alpha <: \mathsf{Int}).\, \alpha \to \alpha$, which, based on the encoding, elaborates to $\forall\alpha.\, \alpha \,\&\, \mathsf{Int} \to \alpha \,\&\, \mathsf{Int}$. The original type expects a type argument which is a subtype of Int; while in the intersection type system, the elaborated type can take any type argument, e.g., Bool, and then expect a term argument of type $\mathsf{Int} \,\&\, \mathsf{Bool}$. In intersection type systems (e.g., [43]) where $\mathsf{Int} \,\&\, \mathsf{Bool}$ is uninhabited (equivalent to the bottom type), $f\,\mathsf{Bool}$ can take nothing. Yet, in calculi with the merge operator, we can have, e.g., $f\,\mathsf{Bool}\,(1 \,,, \,\mathsf{True})$.

7 Related Work

Bounded quantification and intersection types The language Fun [12] introduced bounded quantification. Bounded quantification is later extended with extensible records [10, 11], recursively defined types [9] and session types [25, 33] among other extensions. The full variant of $\mathsf{F}_{<:}$ [23] (see also Section 6.2) is proved to be undecidable [48]. The kernel Fun variant [12], which restricts the subtyping of bounds to be invariant, is decidable.

Pierce [47] proposed the encoding of bounded quantification in terms of intersection types in an informal discussion, which is the main inspiration of our Section 5. Castagna and Xu [19] mentioned in a footnote that a type variable α bounded by a type σ can be encoded by replacing every occurrence of α by $\beta \wedge \sigma$ where β is a fresh unbounded variable. Castagna et al. [17] further mentioned that the possible instantiation of a type variable α with a upper bound σ and a lower bound τ is equivalent to the possible instantiation of $(\tau \vee \beta) \wedge \sigma$. Dolan and Mycroft [26] used a similar encoding as one of the main ingredients of the biunification algorithm: $\alpha <: \sigma^-$ (where types have polarity) implies the bisubstitution $\theta = [(\mu^- \beta.\alpha \sqcap [\beta/\alpha^-](\sigma^-))/\alpha^-]$, which by unrolling implies that $\theta(\alpha^-) = \alpha \sqcap \theta(\sigma^-)$. The idea of encoding bounded quantification using intersection types is not new. However, as far as we know, we are the first to formalize an elaboration and study the metatheory from a calculus with bounded quantification into a calculus with intersection types and polymorphism. This contrasts with the previous informal discussions, which have only shown a few concrete examples of programs that could be manually translated (or not).

Row calculi and intersection types Along the way we have mentioned many row calculi [35, 5, 52, 11, 61, 60]. The typed language *Rose* [42] for extensible data types generalizes various row type systems. *Rose* also generalizes over row type systems with duplicate labels. In contrast, our work is focused on row types with disjoint record types.

Reynolds [55] developed an encoding of simple records in terms of intersection types and his merge construct. Similar ideas had been applied by more recent work on intersection types with a merge operator [28, 6, 2]. Alpuim and Oliveira [2] showed informally that many features of row polymorphism can be simulated with disjoint polymorphism. However, their intersection type system is limiting for the encoding in Section 4.4.

Intersection types and the merge operator The F_i^+ calculus follows from a line of work on intersection types with a merge operator. The programming language Forsythe [55, 53] includes a merge operator. However, several restrictions were imposed to make the merge operator coherent [54]. For example, merging two functions is forbidden. Castagna et al. [14] studied a special merge operator that only works on functions. Dunfield [28] proposed a calculus with unrestricted intersection types and unrestricted merges. However his calculus loses coherence. For example, $1\,,,\,2$ could elaborate to 1 or 2. Pierce [47] proposed a primitive function glue, similar to unrestricted merges. Oliveira et al. [46] proposed disjoint intersection types and disjoint merges to recover syntactic coherence. Later this approach was extended with *disjoint polymorphism* [2]. Bi et al. [6] support unrestricted intersection types and disjoint merges, based on is a novel semantic coherence approach in terms of contextual equivalence, which is later extended to support polymorphic types [7].

Other work on intersection types includes refinement intersections [24, 27]; set theoretical foundation for type connectives including intersections, unions and negations [16, 15, 17, 19]; and the DOT calculus, which aims at providing a foundational calculus for Scala that incorporates features including intersection types [3, 56]. In those calculi, intersection types only increase the expressiveness of types, but not the expressiveness of terms. For example, the intersection type $\mathsf{Int} \,\&\, \mathsf{Bool}$ is uninhabited. The type system of Ceylon [43] exploits this fact and considers any intersection of such *disjoint* types equivalent to the bottom type ($\bot$).

8 Conclusion and Future Work

We have presented the elaboration from kernel $\mathsf{F}_{<:}$ and $\lambda^{||}$ to F_i^+, and showed that disjoint polymorphism is powerful enough to encode essential aspects of bounded quantification and row polymorphism, which is useful for economy of theory and implementation. The elaboration from kernel $\mathsf{F}_{<:}$ identifies one encodable fragment of $\mathsf{F}_{<:}$, and thus validates the previous informal observation by Pierce. The elaboration from $\lambda^{||}$ to F_i^+ reveals the essence of constrained quantification from the point of view of disjointness.

As for future work, we plan to study the encoding of other variants of $\mathsf{F}_{<:}$, e.g., how to combine intersection types with implicit polymorphic subtyping to encode the example $\diamond \vdash \forall(\alpha <: \top).\, \alpha <: \forall(\alpha <: \mathsf{Int}).\, \alpha$; as well as other row calculi, e.g., row type systems in the free group with overwriting extensions. We also plan to study type inference of F_i^+.

References

1 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, 2006.

2 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.

3 Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *Workshop on Foundations of Object-Oriented Languages*, 2012.

4 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.

5 Bernard Berthomieu and Camille Le Monies De Sagazan. A calculus of tagged types, with applications to process languages. *Types for Program Analysis*, page 1, 1995.

6 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The essence of nested composition. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

7 Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive disjoint polymorphism for compositional programming. In *European Symposium on Programming (ESOP)*, 2019.

8 Gilad Bracha. *The programming language jigsaw: mixins, modularity and multiple inheritance.* PhD thesis, Dept. of Computer Science, University of Utah, 1992.

9 Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, volume 89, pages 273–280, 1989.

10 Luca Cardelli. *Extensible records in a pure calculus of subtyping.* Digital. Systems Research Center, 1992.

11 Luca Cardelli and John C Mitchell. Operations on records. In *International Conference on Mathematical Foundations of Programming Semantics*, 1989.

12 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

13 Felice Cardone. Relational semantics for recursive types and bounded quantification. In *International Colloquium on Automata, Languages, and Programming*, pages 164–178. Springer, 1989.

14 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Conference on LISP and Functional Programming*, 1992.

15 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types: part 2: local type inference and type reconstruction. In *Principles of Programming Languages (POPL)*, 2015.

16 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *Principles of Programming Languages (POPL)*, 2014.

17 Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Set-theoretic types for polymorphic variants. In *International Conference on Functional Programming (ICFP)*, 2016.

18 Giuseppe Castagna and Benjamin C Pierce. Decidable bounded quantification. In *Principles of Programming Languages (POPL)*, 1994.

19 Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *International Conference on Functional Programming (ICFP)*, 2011.

20 C. Chambers, D. Ungar, B.W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. 4(3):207–222, 1991.

21 Adriana B Compagnoni and Benjamin C Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science (MSCS)*, 6(5):469–501, 1996.

22 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.

23 Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in f$\leq$. *Mathematical structures in computer science*, 2(1):55–91, 1992.

24 Rowan Davies. *Practical refinement-type checking.* PhD thesis, 2005.

25 Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida. Bounded session types for object oriented languages. In *Formal Methods for Components and Objects*, pages 207–245. Springer, 2007.

26 Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 60–72, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3009837.3009882`, `doi:10.1145/3009837.3009882`.

27 Joshua Dunfield. Refined typechecking with stardust. In *PLPV*, 2007.

28 Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming (JFP)*, 24(2-3):133–165, 2014.

29 Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

30 Erik Ernst. The expression problem, scandinavian style. *On Mechanisms For Specialization*, page 27, 2004.

31 Facebook. Flow. `https://flow.org/`, 2014.

32 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Programming Languages and Systems (APLAS)*, 2006.

33 Simon J Gay. Bounded polymorphism in session types. *Mathematical Structures in Computer Science*, 18(5):895–930, 2008.

34 Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Principles of Programming Languages (POPL)*, 1991.

35 Daan Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 5:297–312, 2005.

36 Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Principles of Programming Languages (POPL)*, 2017.

37 Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pages 91–102. ACM, 2012.

38 Sam Lindley and J Garrett Morris. Lightweight functional session types. *Behavioural Types: from Theory to Tools. River Publishers*, pages 265–286, 2017.

39 Simon Martini. Bounded quantifiers have interval models. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 164–173. ACM, 1988.

40 Microsoft. Typescript. `https://www.typescriptlang.org/`, 2012.

41 Microsoft. `https://www.typescriptlang.org/docs/handbook/advanced-types.html`, 2019. Online; accessed 16 June 2019.

42 J. Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. In *Principles of Programming Languages (POPL)*, 2019.

43 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.

44 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, EPFL, 2004.

45 Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Symposium on Principles of Programming Languages (POPL)*, 1996.

46 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*, 2016.

47 Benjamin C Pierce. *Programming with intersection types and bounded polymorphism.* PhD thesis, University of Pennsylvania, 1991.

48 Benjamin C Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.

49 Benjamin C Pierce and David N Turner. Local type argument synthesis with bounded quantification. Technical report, Technical Report 495, Computer Science Department, Indiana University, 1997.

50 Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577, 1980.

51 Redhat. Ceylon. `https://ceylon-lang.org/`, 2011.

52 Didier Rémy. *Type inference for records in a natural extension of ML*. Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and . . . , 1993.

53 John C Reynolds. Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University, 1988.

54 John C. Reynolds. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*, pages 675–700. Springer Berlin Heidelberg, 1991.

55 John C Reynolds. Design of the programming language forsythe. In *ALGOL-like languages*, pages 173–233. 1997.

56 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

57 Mark Shields and Erik Meijer. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 261–275, New York, NY, USA, 2001. ACM. URL: `http://doi.acm.org/10.1145/360204.360230`, `doi:10.1145/360204.360230`.

58 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Object-oriented Programming: Systems, Languages and Applications (OOPSLA)*, 2012.

59 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

60 Mitchell Wand. Complete type inference for simple objects. In *Symposium on Logic in Computer Science (LICS)*, 1987.

61 Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Symposium on Logic in Computer Science (LICS)*, 1989.

62 Mathhias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Foundations of Object-Oriented Languages*, 2005.

A Full Typing Rules of F_i^+

A.1 Well-formedness

$\boxed{\Delta \vdash A}$ *(Well-formedness of Type)*

$$\textsc{swft-top}\ \frac{}{\Delta \vdash \top} \qquad \textsc{swft-bot}\ \frac{}{\Delta \vdash \bot} \qquad \textsc{swft-nat}\ \frac{}{\Delta \vdash \mathsf{Int}} \qquad \textsc{swft-var}\ \frac{(\alpha * A) \in \Delta}{\Delta \vdash \alpha} \qquad \textsc{swft-arrow}\ \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B}$$

$$\textsc{swft-all}\ \frac{\Delta \vdash A \qquad \Delta, \alpha * A \vdash B}{\Delta \vdash \forall (\alpha * A).\, B} \qquad \textsc{swft-and}\ \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B} \qquad \textsc{swft-rcd}\ \frac{\Delta \vdash A}{\Delta \vdash \{l : A\}}$$

$\boxed{\vdash \Delta}$ *(Well-formedness of Type Context)*

$$\textsc{swfte-empty}\ \frac{}{\vdash \bullet} \qquad \textsc{swfte-var}\ \frac{\vdash \Delta \qquad \Delta \vdash A}{\vdash \Delta, \alpha * A}$$

$\boxed{\Delta \vdash \Gamma}$ *(Well-formedness of Term Context)*

$$\textsc{swfe-empty}\ \frac{}{\Delta \vdash \bullet} \qquad \textsc{swfe-var}\ \frac{\Delta \vdash \Gamma \qquad \Delta \vdash A}{\Delta \vdash \Gamma, x : A}$$

A.2 Disjointness

$\boxed{\rceil A \lceil}$ *(Top-like types)*

$$\textsc{TL-top}\ \frac{}{\rceil \top \lceil} \qquad \textsc{TL-and}\ \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil} \qquad \textsc{TL-arr}\ \frac{\rceil B \lceil}{\rceil A \to B \lceil} \qquad \textsc{TL-rcd}\ \frac{\rceil A \lceil}{\rceil \{l : A\} \lceil} \qquad \textsc{TL-all}\ \frac{\rceil B \lceil}{\rceil \forall (\alpha * A).\, B \lceil}$$

$\boxed{\Delta \vdash A * B}$ *(Disjointness)*

$$\textsc{D-topL}\ \frac{\rceil A \lceil}{\Delta \vdash A * B} \qquad \textsc{D-topR}\ \frac{\rceil B \lceil}{\Delta \vdash A * B} \qquad \textsc{D-arr}\ \frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2} \qquad \textsc{D-andL}\ \frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B}$$

$$\textsc{D-andR}\ \frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2} \qquad \textsc{D-rcdEq}\ \frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}} \qquad \textsc{D-rcdNeq}\ \frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}}$$

$$\textsc{D-tvarL}\ \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B} \qquad \textsc{D-tvarR}\ \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha}$$

$$\textsc{D-forall}\ \frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall (\alpha * A_1).\, B_1 * \forall (\alpha * A_2).\, B_2} \qquad \textsc{D-ax}\ \frac{A *_{ax} B}{\Delta \vdash A * B}$$

$\boxed{A *_{ax} B}$ *(Disjointness axioms)*

$$\textsc{Dax-sym}\ \frac{B *_{ax} A}{A *_{ax} B} \qquad \textsc{Dax-intArr}\ \frac{}{\mathsf{Int} *_{ax} A_1 \to A_2} \qquad \textsc{Dax-intRcd}\ \frac{}{\mathsf{Int} *_{ax} \{l : A\}} \qquad \textsc{Dax-intAll}\ \frac{}{\mathsf{Int} *_{ax} \forall(\alpha * B_1).\, B_2}$$

$$\textsc{Dax-arrAll}\ \frac{}{A_1 \to A_2 *_{ax} \forall(\alpha * B_1).\, B_2} \qquad \textsc{Dax-arrRcd}\ \frac{}{A_1 \to A_2 *_{ax} \{l : B\}} \qquad \textsc{Dax-allRcd}\ \frac{}{\forall(\alpha * A_1).\, A_2 *_{ax} \{l : B\}}$$

A.3 Subtyping

$\boxed{A <: B \rightsquigarrow co}$ *(Declarative subtyping)*

$$\textsc{S-refl}\ \frac{}{A <: A \rightsquigarrow \mathsf{id}} \qquad \textsc{S-trans}\ \frac{A_2 <: A_3 \rightsquigarrow co_1 \qquad A_1 <: A_2 \rightsquigarrow co_2}{A_1 <: A_3 \rightsquigarrow co_1 \circ co_2} \qquad \textsc{S-top}\ \frac{}{A <: \top \rightsquigarrow \mathsf{top}}$$

$$\textsc{S-bot}\ \frac{}{\bot <: A \rightsquigarrow \mathsf{bot}} \qquad \textsc{S-rcd}\ \frac{A <: B \rightsquigarrow co}{\{l : A\} <: \{l : B\} \rightsquigarrow co} \qquad \textsc{S-arr}\ \frac{B_1 <: A_1 \rightsquigarrow co_1 \qquad A_2 <: B_2 \rightsquigarrow co_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow co_1 \to co_2}$$

$$\textsc{S-forall}\ \frac{B_1 <: B_2 \rightsquigarrow co \qquad A_2 <: A_1}{\forall(\alpha * A_1).\, B_1 <: \forall(\alpha * A_2).\, B_2 \rightsquigarrow co_\forall} \qquad \textsc{S-and}\ \frac{A_1 <: A_2 \rightsquigarrow co_1 \qquad A_1 <: A_3 \rightsquigarrow co_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle co_1, co_2 \rangle}$$

$$\textsc{S-andl}\ \frac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1} \qquad \textsc{S-andr}\ \frac{}{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2}$$

$$\textsc{S-distArr}\ \frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3 \rightsquigarrow \mathsf{dist}_\to}$$

$$\textsc{S-distRcd}\ \frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \rightsquigarrow \mathsf{id}}$$

$$\textsc{S-distAll}\ \frac{}{(\forall(\alpha * A).\, B_1) \,\&\, (\forall(\alpha * A).\, B_2) <: \forall(\alpha * A).\, B_1 \,\&\, B_2 \rightsquigarrow \mathsf{dist}_\forall} \qquad \textsc{S-topArr}\ \frac{}{\top <: \top \to \top \rightsquigarrow \mathsf{top}_\to}$$

$$\textsc{S-topRcd}\ \frac{}{\top <: \{l : \top\} \rightsquigarrow \mathsf{id}} \qquad \textsc{S-topAll}\ \frac{}{\top <: \forall(\alpha * \top).\, \top \rightsquigarrow \mathsf{top}_\forall}$$

A.4 Elaboration

$\boxed{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e}$ *(Inference)*

$$\textsc{T-top}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle} \qquad \textsc{T-nat}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \mathsf{Int} \rightsquigarrow i} \qquad \textsc{T-var}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad (x : A) \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\textsc{T-app}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \\ \Delta; \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2\end{array}}{\Delta; \Gamma \vdash E_1\, E_2 \Rightarrow A_2 \rightsquigarrow e_1\, e_2} \qquad \textsc{T-merge}\ \frac{\begin{array}{c}\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \\ \Delta; \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad \Delta \vdash A_1 * A_2\end{array}}{\Delta; \Gamma \vdash E_1 \,,,\, E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow \langle e_1, e_2 \rangle}$$

$$\textsc{T-anno}\ \frac{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Delta;\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e} \qquad \textsc{T-rcd}\ \frac{\Delta;\Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Delta;\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow e} \qquad \textsc{T-proj}\ \frac{\Delta;\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Delta;\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e}$$

$$\textsc{T-tabs}\ \frac{\Delta \vdash A \qquad \Delta, \alpha * A;\Gamma \vdash E \Rightarrow B \rightsquigarrow e}{\Delta;\Gamma \vdash \Lambda(\alpha * A).\, E \Rightarrow \forall(\alpha * A).\, B \rightsquigarrow \Lambda\alpha.\, e}$$

$$\textsc{T-tapp}\ \frac{\Delta;\Gamma \vdash E \Rightarrow \forall(\alpha * B).\, C \rightsquigarrow e \qquad \Delta \vdash A * B}{\Delta;\Gamma \vdash E\, A \Rightarrow [A/\alpha]C \rightsquigarrow e\, |A|}$$

$\boxed{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow e}$ *(Checking)*

$$\textsc{T-abs}\ \frac{\Delta \vdash A \qquad \Delta;\Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Delta;\Gamma \vdash \lambda x.\, E \Leftarrow A \to B \rightsquigarrow \lambda x.\, e} \qquad \textsc{T-sub}\ \frac{\Delta;\Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad B <: A \rightsquigarrow co}{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow co\, e}$$

B Full Typing Rules of F_{co}

B.1 Syntax

Types	τ	$::=$	$\mathsf{Int} \mid \langle\rangle \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \alpha \mid \forall\alpha.\, \tau$
Terms	e	$::=$	$x \mid i \mid \langle\rangle \mid \lambda x.\, e \mid e_1\, e_2 \mid \langle e_1, e_2\rangle \mid \Lambda\alpha.\, e \mid e\, \tau \mid co\, e$
Coercions	co	$::=$	$\mathsf{id} \mid co_1 \circ co_2 \mid \mathsf{top} \mid \mathsf{bot} \mid co_1 \to co_2 \mid \langle co_1, co_2\rangle \mid \pi_1 \mid \pi_2$
		$\mid$	$co_\forall \mid \mathsf{dist}_\to \mid \mathsf{top}_\to \mid \mathsf{top}_\forall \mid \mathsf{dist}_\forall$
Values	v	$::=$	$i \mid \langle\rangle \mid \lambda x.\, e \mid \langle v_1, v_2\rangle \mid \Lambda\alpha.\, e \mid (co_1 \to co_2)\, v \mid co_\forall\, v$
		$\mid$	$\mathsf{dist}_\to\, v \mid \mathsf{top}_\to\, v \mid \mathsf{top}_\forall\, v \mid \mathsf{dist}_\forall\, v$
Term contexts	Ψ	$::=$	$\bullet \mid \Psi, x : \tau$
Type contexts	Φ	$::=$	$\bullet \mid \Phi, \alpha$
Evaluation contexts	$\mathcal{E}$	$::=$	$[\cdot] \mid \mathcal{E}\, e \mid v\, \mathcal{E} \mid \langle \mathcal{E}, e\rangle \mid \langle v, \mathcal{E}\rangle \mid co\, \mathcal{E} \mid \mathcal{E}\, \tau$

B.2 Semantics

$\boxed{\Phi \vdash \Psi}$ *(Well-formedness of value context)*

$$\textsc{wfe-empty}\ \frac{}{\Phi \vdash \bullet} \qquad \textsc{wfe-var}\ \frac{\Phi \vdash \tau \qquad \Phi \vdash \Psi \qquad x \notin \Psi \qquad x \notin \Phi}{\Phi \vdash \Psi, x : \tau}$$

$\boxed{\Phi \vdash \tau}$ *(Well-formedness of types)*

$$\textsc{wft-unit}\ \frac{}{\Phi \vdash \langle\rangle} \qquad \textsc{wft-nat}\ \frac{}{\Phi \vdash \mathsf{Int}} \qquad \textsc{wft-var}\ \frac{\alpha \in \Phi}{\Phi \vdash \alpha} \qquad \textsc{wft-arrow}\ \frac{\Phi \vdash \tau_1 \qquad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \to \tau_2} \qquad \textsc{wft-prod}\ \frac{\Phi \vdash \tau_1 \qquad \Phi \vdash \tau_2}{\Phi \vdash \tau_1 \times \tau_2}$$

$$\textsc{wft-all}\ \frac{\Phi, \alpha \vdash \tau_2}{\Phi \vdash \forall\alpha.\, \tau_2}$$

$\boxed{co :: \tau_1 \rhd \tau_2}$ *(Coercion typing)*

$$\textsc{CT-Refl}\ \frac{}{\mathsf{id} :: \tau \rhd \tau} \qquad \textsc{CT-Trans}\ \frac{co_1 :: \tau_2 \rhd \tau_3 \qquad co_2 :: \tau_1 \rhd \tau_2}{co_1 \circ co_2 :: \tau_1 \rhd \tau_3} \qquad \textsc{CT-Top}\ \frac{}{\mathsf{top} :: \tau \rhd \langle\rangle} \qquad \textsc{CT-Bot}\ \frac{}{\mathsf{bot} :: \forall \alpha.\, \alpha \rhd \tau}$$

$$\textsc{CT-TopArr}\ \frac{}{\mathsf{top}_{\rightarrow} :: \langle\rangle \rhd \langle\rangle \rightarrow \langle\rangle} \qquad \textsc{CT-Arr}\ \frac{co_1 :: \tau_1' \rhd \tau_1 \qquad co_2 :: \tau_2 \rhd \tau_2'}{co_1 \rightarrow co_2 :: \tau_1 \rightarrow \tau_2 \rhd \tau_1' \rightarrow \tau_2'} \qquad \textsc{CT-Pair}\ \frac{co_1 :: \tau_1 \rhd \tau_2 \qquad co_2 :: \tau_1 \rhd \tau_3}{\langle co_1, co_2 \rangle :: \tau_1 \rhd \tau_2 \times \tau_3}$$

$$\textsc{CT-DistArr}\ \frac{}{\mathsf{dist}_{\rightarrow} :: (\tau_1 \rightarrow \tau_2) \times (\tau_1 \rightarrow \tau_3) \rhd \tau_1 \rightarrow \tau_2 \times \tau_3} \qquad \textsc{CT-DistAll}\ \frac{}{\mathsf{dist}_{\forall} :: (\forall \alpha.\, \tau_1) \times (\forall \alpha.\, \tau_2) \rhd \forall \alpha.\, \tau_1 \times \tau_2}$$

$$\textsc{CT-ProjL}\ \frac{}{\pi_1 :: \tau_1 \times \tau_2 \rhd \tau_1} \qquad \textsc{CT-ProjR}\ \frac{}{\pi_2 :: \tau_1 \times \tau_2 \rhd \tau_2} \qquad \textsc{CT-Forall}\ \frac{co :: \tau_1 \rhd \tau_2}{co_{\forall} :: \forall \alpha.\, \tau_1 \rhd \forall \alpha.\, \tau_2} \qquad \textsc{CT-TopAll}\ \frac{}{\mathsf{top}_{\forall} :: \langle\rangle \rhd \forall \alpha.\, \langle\rangle}$$

$\boxed{\Phi; \Psi \vdash e : \tau}$ *(Static semantics)*

$$\textsc{T-Unit}\ \frac{\Phi \vdash \Psi}{\Phi; \Psi \vdash \langle\rangle : \langle\rangle} \qquad \textsc{T-Nat}\ \frac{\Phi \vdash \Psi}{\Phi; \Psi \vdash i : \mathsf{Int}} \qquad \textsc{T-Var}\ \frac{\Phi \vdash \Psi \qquad (x : \tau) \in \Psi}{\Phi; \Psi \vdash x : \tau}$$

$$\textsc{T-Abs}\ \frac{\Phi; \Psi, x : \tau_1 \vdash e : \tau_2 \qquad \Phi \vdash \tau_1}{\Phi; \Psi \vdash \lambda x.\, e : \tau_1 \rightarrow \tau_2} \qquad \textsc{T-App}\ \frac{\Phi; \Psi \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Phi; \Psi \vdash e_2 : \tau_1}{\Phi; \Psi \vdash e_1\ e_2 : \tau_2}$$

$$\textsc{T-TAbs}\ \frac{\Phi, \alpha; \Psi \vdash e : \tau \qquad \Phi \vdash \Psi}{\Phi; \Psi \vdash \Lambda \alpha.\, e : \forall \alpha.\, \tau} \qquad \textsc{T-TApp}\ \frac{\Phi; \Psi \vdash e : \forall \alpha.\, \tau' \qquad \Phi \vdash \tau}{\Phi; \Psi \vdash e\, \tau : [\tau/\alpha]\tau'}$$

$$\textsc{T-Pair}\ \frac{\Phi; \Psi \vdash e_1 : \tau_1 \qquad \Phi; \Psi \vdash e_2 : \tau_2}{\Phi; \Psi \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \textsc{T-CApp}\ \frac{\Phi; \Psi \vdash e : \tau \qquad co :: \tau \rhd \tau'}{\Phi; \Psi \vdash co\, e : \tau'}$$

$\boxed{e \longrightarrow e'}$ *(Single-step reduction)*

$$\textsc{R-TopArr}\ \frac{}{(\mathsf{top}_{\rightarrow}\, \langle\rangle)\, \langle\rangle \longrightarrow \langle\rangle} \qquad \textsc{R-TopAll}\ \frac{}{(\mathsf{top}_{\forall}\, \langle\rangle)\, \tau \longrightarrow \langle\rangle} \qquad \textsc{R-DistArr}\ \frac{}{(\mathsf{dist}_{\rightarrow}\, \langle v_1, v_2 \rangle)\, v_3 \longrightarrow \langle v_1\, v_3, v_2\, v_3 \rangle}$$

$$\textsc{R-DistAll}\ \frac{}{(\mathsf{dist}_{\forall}\, \langle v_1, v_2 \rangle)\, \tau \longrightarrow \langle v_1\, \tau, v_2\, \tau \rangle} \qquad \textsc{R-Id}\ \frac{}{\mathsf{id}\, v \longrightarrow v} \qquad \textsc{R-Trans}\ \frac{}{(co_1 \circ co_2)\, v \longrightarrow co_1\, (co_2\, v)}$$

$$\textsc{R-Top}\ \frac{}{\mathsf{top}\, v \longrightarrow \langle\rangle} \qquad \textsc{R-Arr}\ \frac{}{((co_1 \rightarrow co_2)\, v_1)\, v_2 \longrightarrow co_2\, (v_1\, (co_1\, v_2))} \qquad \textsc{R-Pair}\ \frac{}{\langle co_1, co_2 \rangle\, v \longrightarrow \langle co_1\, v, co_2\, v \rangle}$$

$$\textsc{R-ProjL}\ \frac{}{\pi_1\, \langle v_1, v_2 \rangle \longrightarrow v_1} \qquad \textsc{R-ProjR}\ \frac{}{\pi_2\, \langle v_1, v_2 \rangle \longrightarrow v_2} \qquad \textsc{R-Forall}\ \frac{}{(co_{\forall}\, v)\, \tau \longrightarrow co\, (v\, \tau)} \qquad \textsc{R-App}\ \frac{}{(\lambda x.\, e)\, v \longrightarrow [v/x]e}$$

$$\textsc{R-TApp}\ \frac{}{(\Lambda \alpha.\, e)\, \tau \longrightarrow [\tau/\alpha]e} \qquad \textsc{R-Ctxt}\ \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

C Full Typing Rules of $\lambda^{||}$

C.1 Syntax

Types	t	$::=$	$\mathsf{Int} \mid t_1 \to t_2 \mid \forall \alpha \# R.\, t \mid r$
Records	r	$::=$	$\alpha \mid \mathsf{Empty} \mid \{l : t\} \mid r_1 \mathbin{\|\|} r_2$
Constraint lists	R	$::=$	$\diamond \mid r, R$
Terms	ε	$::=$	$x \mid \lambda(x : t).\, \varepsilon \mid \varepsilon_1\, \varepsilon_2 \mid \mathsf{empty} \mid \{l = \varepsilon\} \mid \varepsilon_1 \mathbin{\|\|} \varepsilon_2$
		$\mid$	$\varepsilon \setminus l \mid \varepsilon.l \mid \Lambda(\alpha \# R).\, \varepsilon \mid \varepsilon\ [r]$
Term contexts	G	$::=$	$\diamond \mid G, x : t$
Type contexts	T	$::=$	$\diamond \mid T, \alpha \# R$

C.2 Semantics

$\boxed{T\ \mathsf{ok}}$ *(Well-formed type contexts)*

$$\textsc{wftc-Empty}\quad \frac{}{\diamond\ \mathsf{ok}} \qquad\qquad \textsc{wftc-TVar}\quad \frac{T\ \mathsf{ok} \qquad T \vdash R\ \mathsf{ok}}{T, \alpha \# R\ \mathsf{ok}}$$

$\boxed{T \vdash G\ \mathsf{ok}}$ *(Well-formed term contexts)*

$$\textsc{wfc-Empty}\quad \frac{T\ \mathsf{ok}}{T \vdash \diamond\ \mathsf{ok}} \qquad\qquad \textsc{wfc-Var}\quad \frac{T \vdash G\ \mathsf{ok} \qquad T \vdash t\ \mathsf{type}}{T \vdash G, x : t\ \mathsf{ok}}$$

$\boxed{T \vdash t\ \mathsf{type}}$ *(Well-formed types)*

$$\textsc{wfrt-Prim}\quad \frac{}{T \vdash \mathsf{Int}\ \mathsf{type}} \qquad \textsc{wfrt-Arrow}\quad \frac{T \vdash t_1\ \mathsf{type} \qquad T \vdash t_2\ \mathsf{type}}{T \vdash t_1 \to t_2\ \mathsf{type}} \qquad \textsc{wfrt-All}\quad \frac{T \vdash R\ \mathsf{ok} \qquad T, \alpha \# R \vdash t\ \mathsf{type}}{T \vdash \forall \alpha \# R.\, t\ \mathsf{type}}$$

$$\textsc{wfrt-Rec}\quad \frac{T \vdash r\ \mathsf{record}}{T \vdash r\ \mathsf{type}}$$

$\boxed{T \vdash r\ \mathsf{record}}$ *(Well-formed record types)*

$$\textsc{wfr-Var}\quad \frac{(\alpha \# R) \in T}{T \vdash \alpha\ \mathsf{record}} \qquad \textsc{wfr-Base}\quad \frac{T \vdash t\ \mathsf{type}}{T \vdash \{l : t\}\ \mathsf{record}} \qquad \textsc{wfr-Empty}\quad \frac{}{T \vdash \mathsf{Empty}\ \mathsf{record}}$$

$$\textsc{wfr-Merge}\quad \frac{T \vdash r_1\ \mathsf{record} \qquad T \vdash r_2\ \mathsf{record} \qquad T \vdash r_1 \# r_2}{T \vdash r_1 \mathbin{\|\|} r_2\ \mathsf{record}}$$

$\boxed{T \vdash R\ \mathsf{ok}}$ *(Well-formed constraint lists)*

$$\textsc{wfcl-Nil}\quad \frac{T\ \mathsf{ok}}{T \vdash \diamond\ \mathsf{ok}} \qquad\qquad \textsc{wfcl-Cons}\quad \frac{T \vdash r\ \mathsf{record} \qquad T \vdash R\ \mathsf{ok}}{T \vdash r, R\ \mathsf{ok}}$$

$\boxed{T \vdash r_1 \# r_2}$ *(Compatibility)*

$$\textsc{cmp-Eq}\ \frac{T \vdash r \# s \qquad r \sim r' \qquad s \sim s'}{T \vdash r' \# s'} \qquad \textsc{cmp-Symm}\ \frac{T \vdash r \# s}{T \vdash s \# r} \qquad \textsc{cmp-Base}\ \frac{T \vdash r \# \{l : t\} \qquad T \vdash t' \ \textsf{type}}{T \vdash r \# \{l : t'\}}$$

$$\textsc{cmp-TVar}\ \frac{(\alpha \# R) \in T \qquad T \vdash R \ \textsf{ok} \qquad r \in R}{T \vdash \alpha \# r} \qquad \textsc{cmp-MergeE}\ \frac{T \vdash r \# (s_1 \mid\mid s_2)}{T \vdash r \# s_i}$$

$$\textsc{cmp-MergeI}\ \frac{T \vdash s_1 \# s_2 \qquad T \vdash r \# s_1 \qquad T \vdash r \# s_2}{T \vdash r \# (s_1 \mid\mid s_2)} \qquad \textsc{cmp-BaseBase}\ \frac{l \neq l' \qquad T \vdash t \ \textsf{type} \qquad T \vdash t' \ \textsf{type}}{T \vdash \{l : t\} \# \{l' : t'\}}$$

$$\textsc{cmp-Empty}\ \frac{T \vdash r \ \textsf{record}}{T \vdash r \# \textsf{Empty}}$$

$\boxed{T \vdash r \# R}$ *(Constraint list satisfaction)*

$$\textsc{cmpList-Nil}\ \frac{T \vdash r \ \textsf{record}}{T \vdash r \# \diamond} \qquad \textsc{cmpList-Cons}\ \frac{T \vdash r \# r' \qquad T \vdash r \# R}{T \vdash r \# r', R}$$

$\boxed{t_1 \sim t_2}$ *(Type equivalence)*

$$\textsc{teq-Refl}\ \frac{}{t \sim t} \qquad \textsc{teq-Symm}\ \frac{t \sim t'}{t' \sim t} \qquad \textsc{teq-Trans}\ \frac{t \sim t' \qquad t' \sim t''}{t \sim t''} \qquad \textsc{teq-CongArrow}\ \frac{t_1 \sim t_1' \qquad t_2 \sim t_2'}{t_1 \to t_2 \sim t_1' \to t_2'}$$

$$\textsc{teq-CongAll}\ \frac{R \sim R' \qquad t \sim t'}{\forall \alpha \# R. t \sim \forall \alpha \# R'. t'} \qquad \textsc{teq-CongBase}\ \frac{t \sim t'}{\{l : t\} \sim \{l : t'\}} \qquad \textsc{teq-CongMerge}\ \frac{r_1 \sim r_1' \qquad r_2 \sim r_2'}{r_1 \mid\mid r_2 \sim r_1' \mid\mid r_2'} \qquad \textsc{teq-MergeUnit}\ \frac{}{r \mid\mid \textsf{Empty} \sim r}$$

$$\textsc{teq-MergeAssoc}\ \frac{}{r_1 \mid\mid (r_2 \mid\mid r_3) \sim (r_1 \mid\mid r_2) \mid\mid r_3} \qquad \textsc{teq-MergeComm}\ \frac{}{r_1 \mid\mid r_2 \sim r_2 \mid\mid r_1}$$

$\boxed{R_1 \sim R_2}$ *(Constraint list equivalence)*

$$\textsc{ceq-Refl}\ \frac{}{R \sim R} \qquad \textsc{ceq-Symm}\ \frac{R \sim R'}{R' \sim R} \qquad \textsc{ceq-Trans}\ \frac{R \sim R' \qquad R' \sim R''}{R \sim R''} \qquad \textsc{ceq-Inner}\ \frac{R \sim R' \qquad r \sim r'}{r, R \sim r', R'}$$

$$\textsc{ceq-Swap}\ \frac{}{r, (r', R) \sim r', (r, R)} \qquad \textsc{ceq-Empty}\ \frac{}{\textsf{Empty}, R \sim R} \qquad \textsc{ceq-Merge}\ \frac{}{(r_1 \mid\mid r_2), R \sim r_1, (r_2, R)} \qquad \textsc{ceq-Dupl}\ \frac{}{r, (r, R) \sim r, R}$$

$$\textsc{ceq-Base}\ \frac{}{\{l : t\}, R \sim \{l : t'\}, R}$$

C.3 Intuitive Elaboration

$\boxed{T; G \vdash \varepsilon : t \leadsto_i E}$ *(Type-directed translation)*

$$\textsc{wtti-Eq}\quad \frac{T;G\vdash\varepsilon:t\rightsquigarrow_i E \qquad T\vdash t'\ \mathsf{type} \qquad t\sim t'}{T;G\vdash\varepsilon:t'\rightsquigarrow_i E:[\![t']\!]}$$

$$\textsc{wtti-Var}\quad \frac{T\ \mathsf{ok} \qquad T\vdash G\ \mathsf{ok} \qquad (x:t)\in G}{T;G\vdash x:t\rightsquigarrow_i x}$$

$$\textsc{wtti-Nat}\quad \frac{T\ \mathsf{ok} \qquad T\vdash G\ \mathsf{ok}}{T;G\vdash i:\mathsf{Int}\rightsquigarrow_i i}$$

$$\textsc{wtti-ArrowI}\quad \frac{T\vdash t\ \mathsf{type} \qquad T;G,x:t\vdash\varepsilon:t'\rightsquigarrow_i E}{T;G\vdash\lambda(x:t).\,\varepsilon:t\to t'\rightsquigarrow_i(\lambda x.\,E):[\![t]\!]\to[\![t']\!]}$$

$$\textsc{wtti-ArrowE}\quad \frac{T;G\vdash\varepsilon_1:t\to t'\rightsquigarrow_i E_1 \qquad T;G\vdash\varepsilon_2:t\rightsquigarrow_i E_2}{T;G\vdash\varepsilon_1\,\varepsilon_2:t'\rightsquigarrow_i E_1\,E_2}$$

$$\textsc{wtti-Base}\quad \frac{T;G\vdash\varepsilon:t\rightsquigarrow_i E}{T;G\vdash\{l=\varepsilon\}:\{l:t\}\rightsquigarrow_i\{l=E\}}$$

$$\textsc{wtti-Restr}\quad \frac{T;G\vdash\varepsilon:\{l:t\}\mid\mid r\rightsquigarrow_i E}{T;G\vdash\varepsilon\setminus l:r\rightsquigarrow_i E:[\![r]\!]}$$

$$\textsc{wtti-Empty}\quad \frac{T\ \mathsf{ok} \qquad T\vdash G\ \mathsf{ok}}{T;G\vdash\mathsf{empty}:\mathsf{Empty}\rightsquigarrow_i\top}$$

$$\textsc{wtti-Merge}\quad \frac{\begin{array}{c}T;G\vdash\varepsilon_1:r_1\rightsquigarrow_i E_1\\ T;G\vdash\varepsilon_2:r_2\rightsquigarrow E_2 \qquad T\vdash r_1\,\#\,r_2\end{array}}{T;G\vdash\varepsilon_1\mid\mid\varepsilon_2:r_1\mid\mid r_2\rightsquigarrow_i E_1\,,,\,E_2}$$

$$\textsc{wtti-Select}\quad \frac{T;G\vdash\varepsilon:\{l:t\}\mid\mid r\rightsquigarrow_i E}{T;G\vdash\varepsilon.l:t\rightsquigarrow_i(E:\{l:[\![t]\!]\}).l}$$

$$\textsc{wtti-AllI}\quad \frac{T\vdash R\ \mathsf{ok} \qquad T,\alpha\,\#\,R;G\vdash\varepsilon:t\rightsquigarrow_i E}{T;G\vdash\Lambda(\alpha\,\#\,R).\,\varepsilon:\forall\alpha\,\#\,R.\,t\rightsquigarrow_i\Lambda(\alpha*[\![R]\!]).\,E}$$

$$\textsc{wtti-AllE}\quad \frac{T;G\vdash\varepsilon:\forall\alpha\,\#\,R.\,t\rightsquigarrow_i E \qquad T\vdash r\,\#\,R}{T;G\vdash\varepsilon\ [r]:[r/\alpha]t\rightsquigarrow_i E\,[\![r]\!]}$$

C.4 Full Elaboration

$$\boxed{T;G\vdash\varepsilon:t\rightsquigarrow E}$$ *(Type-directed translation)*

$$\textsc{wtt-Eq}\quad \frac{T;G\vdash\varepsilon:t\rightsquigarrow E \qquad T\vdash t'\ \mathsf{type} \qquad t\sim t'}{T;G\vdash\varepsilon:t'\rightsquigarrow E:[\![t']\!]}$$

$$\textsc{wtt-Var}\quad \frac{T\ \mathsf{ok} \qquad T\vdash G\ \mathsf{ok} \qquad (x:t)\in G}{T;G\vdash x:t\rightsquigarrow x}$$

$$\textsc{wtt-Nat}\quad \frac{T\ \mathsf{ok} \qquad T\vdash G\ \mathsf{ok}}{T;G\vdash i:\mathsf{Int}\rightsquigarrow i}$$

$$\textsc{wtt-ArrowI}\quad \frac{T\vdash t\ \mathsf{type} \qquad T;G,x:t\vdash\varepsilon:t'\rightsquigarrow E}{T;G\vdash\lambda(x:t).\,\varepsilon:t\to t'\rightsquigarrow(\lambda x.\,E):[\![t]\!]\to[\![t']\!]}$$

$$\textsc{wtt-ArrowE}\quad \frac{T;G\vdash\varepsilon_1:t\to t'\rightsquigarrow E_1 \qquad T;G\vdash\varepsilon_2:t\rightsquigarrow E_2}{T;G\vdash\varepsilon_1\,\varepsilon_2:t'\rightsquigarrow E_1\,E_2}$$

$$\textsc{wtt-Base}\quad \frac{T;G\vdash\varepsilon:t\rightsquigarrow E}{T;G\vdash\{l=\varepsilon\}:\{l:t\}\rightsquigarrow\{l=E\}}$$

$$\textsc{wtt-Restr}\quad \frac{T;G\vdash\varepsilon:\{l:t\}\mid\mid r\rightsquigarrow E}{T;G\vdash\varepsilon\setminus l:r\rightsquigarrow E:[\![r]\!]}$$

$$\textsc{wtt-Empty}\quad \frac{T\ \mathsf{ok} \qquad T\vdash G\ \mathsf{ok}}{T;G\vdash\mathsf{empty}:\mathsf{Empty}\rightsquigarrow\top}$$

$$\textsc{wtt-Merge}\quad \frac{\begin{array}{c}T;G\vdash\varepsilon_1:r_1\rightsquigarrow E_1\\ T;G\vdash\varepsilon_2:r_2\rightsquigarrow E_2 \qquad T\vdash r_1\,\#\,r_2\end{array}}{T;G\vdash\varepsilon_1\mid\mid\varepsilon_2:r_1\mid\mid r_2\rightsquigarrow E_1\,,,\,E_2}$$

$$\textsc{wtt-Select}\quad \frac{T;G\vdash\varepsilon:\{l:t\}\mid\mid r\rightsquigarrow E}{T;G\vdash\varepsilon.l:t\rightsquigarrow(E:\{l:[\![t]\!]\}).l}$$

$$\textsc{wtt-AllI}\quad \frac{T \vdash R\ \mathsf{ok} \qquad T, \alpha \# R; G \vdash \varepsilon : t \rightsquigarrow E}{T; G \vdash \Lambda(\alpha \# R).\, \varepsilon : \forall \alpha \# R.\, t \rightsquigarrow \Lambda(\alpha * \llbracket R \rrbracket).\, \Lambda(\alpha_{\perp} * \llbracket R \rrbracket).\, E}$$

$$\textsc{wtt-AllE}\quad \frac{T; G \vdash \varepsilon : \forall \alpha \# R.\, t \rightsquigarrow E \qquad T \vdash r \# R}{T; G \vdash \varepsilon\ [r] : [r/\alpha]t \rightsquigarrow E\, \llbracket r \rrbracket\, \llbracket r \rrbracket_{\perp}}$$

D Kernel $\mathsf{F}_{<:}$

$\boxed{\epsilon_1 \longrightarrow \epsilon_2}$ *(Reduction)*

$$\textsc{f-red-app1}\quad \frac{\epsilon_1 \longrightarrow \epsilon_1'}{\epsilon_1\, \epsilon_2 \longrightarrow \epsilon_1'\, \epsilon_2} \qquad \textsc{f-red-app2}\quad \frac{\epsilon_2 \longrightarrow \epsilon_2'}{\upsilon\, \epsilon_2 \longrightarrow \upsilon\, \epsilon_2'} \qquad \textsc{f-red-app}\quad \frac{}{(\lambda(x : \sigma_1).\, \epsilon)\, \upsilon \longrightarrow [\upsilon/x]\epsilon} \qquad \textsc{f-red-tapp1}\quad \frac{\epsilon \longrightarrow \epsilon'}{\epsilon\, \tau \longrightarrow \epsilon'\, \tau}$$

$$\textsc{f-red-tapp}\quad \frac{}{(\Lambda(\alpha <: \sigma).\, \epsilon)\, \tau \longrightarrow [\tau/\alpha]\epsilon}$$

$$\textsc{f-red-rcd}\quad \frac{\epsilon_i \longrightarrow \epsilon_i'}{\{l_1 = \epsilon_1, \mathinner{..}, l_i = \epsilon_i, \mathinner{..}, l_n = \epsilon_n\} \longrightarrow \{l_1 = \epsilon_1, \mathinner{..}, l_i = \epsilon_i', \mathinner{..}, l_n = \epsilon_n\}}$$

$$\textsc{f-red-proj}\quad \frac{}{\{l_1 = \upsilon_1, \mathinner{..}, l_i = \upsilon_i, \mathinner{..}, l_n = \upsilon_n\}.l \longrightarrow \upsilon_i}$$

▶ **Lemma 22** (Elaboration substitution of term variables). *If* $\Sigma_1, x : \sigma, \Sigma_2 \vdash \epsilon_1 : \tau \rightsquigarrow E_1$, *and* $\Sigma_1, \Sigma_2 \vdash \epsilon_2 : \sigma \rightsquigarrow E_2$, *then* $\Sigma_1, \Sigma_2 \vdash [\epsilon_2/x]\epsilon_1 : \sigma \rightsquigarrow [E_2/x]E_1$.

Proof. Follows by induction on the typing judgment. ◀

The following lemma needs to generalize the heterogeneous logical relation of F_i^+ to have two term contexts.

▶ **Lemma 23** (Elaboration substitution of type variables). *If* $\Sigma_1, \alpha <: \sigma_1, \Sigma_2 \vdash \epsilon_1 : \tau \rightsquigarrow E_1$, *and* $\Sigma_1 \vdash \sigma_2 <: \sigma_1$, *then* $\Sigma_1, ([\sigma_2/\alpha]\Sigma_2) \vdash [\sigma_2/\alpha]\epsilon : [\sigma_2/\alpha]\tau \rightsquigarrow E_2$.
Moreover, if $\lfloor\!\lfloor \Sigma_1, \alpha <: \sigma_1, \Sigma_2 \rfloor\!\rfloor; \lceil\!\lceil \Sigma_1, \alpha <: \sigma_1, \Sigma_2 \rceil\!\rceil \vdash E_1 \Rightarrow \llbracket \tau \rrbracket_{(\Sigma_1, \alpha <: \sigma_1, \Sigma_2)} \rightsquigarrow e_1$,
and $\lfloor\!\lfloor \Sigma_1, ([\sigma_2/\alpha]\Sigma_2) \rfloor\!\rfloor; \lceil\!\lceil \Sigma_1, ([\sigma_2/\alpha]\Sigma_2) \rceil\!\rceil \vdash E_2 \Rightarrow \llbracket ([\sigma_2/\alpha]\tau) \rrbracket_{(\Sigma_1, ([\sigma_2/\alpha]\Sigma_2))} \rightsquigarrow e_2$,
then $\lfloor\!\lfloor \Sigma_1, ([\sigma_2/\alpha]\Sigma_2) \rfloor\!\rfloor; \lceil\!\lceil \Sigma_1, ([\sigma_2/\alpha]\Sigma_2) \rceil\!\rceil; [\llbracket \sigma_2 \rrbracket_{\Sigma_1}/\alpha] \lceil\!\lceil \Sigma_1, \alpha <: \sigma_1, \Sigma_2 \rceil\!\rceil$
$\vdash e_2 \backsimeq_{log} [|\llbracket \sigma_2 \rrbracket_{\Sigma_1}|/\alpha] e_1 : \llbracket ([\sigma_2/\alpha]\tau) \rrbracket_{(\Sigma_1, ([\sigma_2/\alpha]\Sigma_2))}; [\llbracket \sigma_2 \rrbracket_{\Sigma_1}/\alpha](\llbracket \tau \rrbracket_{(\Sigma_1, \alpha <: \sigma_1, \Sigma_2)})$.

Proof. The first half follows by induction on the typing judgment. Note that, E_2 does not have a substitution relation with E_1.

The second half is more involved. The key observation is that, when we substitute a type variable, the only change in E's are type annotations, which are coercions in e's. According to Lemma 18, we always get isomorphic types. So we can prove that the results are logically equivalent according to the compatibility of coercions. ◀

▶ **Lemma 24** (Generalized coherence of operational semantics). *If* $\epsilon_1 \longrightarrow \epsilon_2$, *and* $\diamond \vdash \epsilon_1 : \sigma \rightsquigarrow E_1$, *and* $\bullet; \bullet \vdash E_1 \Rightarrow \llbracket \sigma \rrbracket_{\diamond} \rightsquigarrow e_1$, *then* $\diamond \vdash \epsilon_2 : \sigma \rightsquigarrow E_2$, *and* $\bullet; \bullet \vdash E_2 \Rightarrow \llbracket \sigma \rrbracket_{\diamond} \rightsquigarrow e_2$, *then* $\bullet; \bullet \vdash e_1 \backsimeq_{log} e_2 : \llbracket \sigma \rrbracket_{\diamond}$.

Proof. By induction on the reduction judgment.

- Case

F-RED-APP1
$$\frac{\epsilon_1 \longrightarrow \epsilon_1'}{\epsilon_1\,\epsilon_2 \longrightarrow \epsilon_1'\,\epsilon_2}$$

$\diamond \vdash \epsilon_1\,\epsilon_2 : \sigma \rightsquigarrow E_1$	Given
$\bullet;\bullet \vdash E_1 \Rightarrow \llbracket\sigma\rrbracket_\diamond \rightsquigarrow e_1$	Given
$\diamond \vdash \epsilon_1 : \tau \to \sigma_1 \rightsquigarrow E_2$	By inversion
$\diamond \vdash \epsilon_2 : \tau \rightsquigarrow E_3$	
$\diamond \vdash \sigma_1 <: \sigma$	
$\bullet \vdash \llbracket\sigma_1\rrbracket_\diamond <: \llbracket\sigma\rrbracket_\diamond \rightsquigarrow co$	
$\bullet;\bullet \vdash E_2 \Rightarrow \llbracket\tau \to \sigma_1\rrbracket_\diamond \rightsquigarrow e_2$	
$\bullet;\bullet \vdash E_3 \Leftarrow \llbracket\tau\rrbracket_\diamond \rightsquigarrow e_3$	
$\bullet;\bullet \vdash e_1 \backsimeq_{log} e_2\,e_3 : \llbracket\sigma\rrbracket_\diamond; \llbracket\sigma_1\rrbracket_\diamond$	By compatibility of coercions
$\diamond \vdash \epsilon_1' : \tau \to \sigma_1 \rightsquigarrow E_4$	I.H.
$\bullet;\bullet \vdash E_4 \Rightarrow \llbracket\tau \to \sigma_1\rrbracket_\diamond \rightsquigarrow e_4$	
$\bullet;\bullet \vdash e_2 \backsimeq_{log} e_4 : \llbracket\tau \to \sigma_1\rrbracket_\diamond$	
$\diamond \vdash \epsilon_1'\,\epsilon_2 : \sigma_1 \rightsquigarrow E_4\,E_3$	By rule F-APP
$\diamond \vdash \epsilon_1'\,\epsilon_2 : \sigma \rightsquigarrow (E_4\,E_3) : \llbracket\sigma\rrbracket_\diamond$	By rule F-SUB
$\bullet;\bullet \vdash (E_4\,E_3) : \llbracket\sigma\rrbracket_\diamond \Rightarrow \llbracket\sigma\rrbracket_\diamond \rightsquigarrow co\,(e_4\,e_3)$	By rule T-SUB
$\bullet;\bullet \vdash e_2\,e_3 \backsimeq_{log} e_4\,e_3 : \llbracket\sigma_1\rrbracket_\diamond$	By compatibility on application
$\bullet;\bullet \vdash e_2\,e_3 \backsimeq_{log} co\,(e_4\,e_3) : \llbracket\sigma_1\rrbracket_\diamond; \llbracket\sigma\rrbracket_\diamond$	By compatibility on coercions
$\bullet;\bullet \vdash e_1 \backsimeq_{log} e_4\,e_3 : \llbracket\sigma\rrbracket_\diamond$	By transitivity

- The case for rule F-RED-APP2, rule F-RED-TAPP1 and rule F-RED-RCD are similar to the previous case. In the latter two cases, instead of using rules and compatibility on application, we use rules and compatibility on type applications, and merges, respectively.
- Case

F-RED-APP
$$\frac{}{(\lambda(x : \sigma_1).\,\epsilon)\,\upsilon \longrightarrow [\upsilon/x]\epsilon}$$

$\diamond \vdash (\lambda(x : \sigma_1).\,\epsilon)\,\upsilon : \sigma \rightsquigarrow E_1$	Given
$\bullet;\bullet \vdash E_1 \Rightarrow \llbracket\sigma\rrbracket_\diamond \rightsquigarrow e_1$	Given
$\diamond \vdash (\lambda(x : \sigma_1).\,\epsilon) : \tau_1 \to \tau_2 \rightsquigarrow E_2$	By inversion
$\diamond \vdash \upsilon : \tau_1 \rightsquigarrow E_3$	
$\diamond \vdash \tau_2 <: \sigma$	
$\bullet \vdash \llbracket\tau_2\rrbracket_\diamond <: \llbracket\sigma\rrbracket_\diamond \rightsquigarrow co$	
$\bullet;\bullet \vdash E_2 \Rightarrow \llbracket\tau_1 \to \tau_2\rrbracket_\diamond \rightsquigarrow e_2$	
$\bullet;\bullet \vdash E_3 \Leftarrow \llbracket\tau_1\rrbracket_\diamond \rightsquigarrow e_3$	
$\bullet;\bullet \vdash e_1 \backsimeq_{log} e_2\,e_3 : \llbracket\sigma\rrbracket_\diamond; \llbracket\tau_2\rrbracket_\diamond$	By compatibility of coercions
$\diamond \vdash (\lambda(x : \sigma_1).\,\epsilon) : \sigma_1 \to \sigma_2 \rightsquigarrow (\lambda x.\,E_4) : \llbracket\sigma_1 \to \sigma_2\rrbracket_\diamond$	By inversion
$\bullet;\bullet \vdash (\lambda x.\,E_4) : \llbracket\sigma_1 \to \sigma_2\rrbracket_\diamond \Rightarrow \llbracket\sigma_1 \to \sigma_2\rrbracket_\diamond \rightsquigarrow (\lambda x.\,e_4)$	
$\diamond \vdash \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2$	
$\bullet \vdash \llbracket\sigma_1 \to \sigma_2\rrbracket_\diamond <: \llbracket\tau_1 \to \tau_2\rrbracket_\diamond \rightsquigarrow co_1$	
$\bullet;\bullet \vdash e_2 \backsimeq_{log} (\lambda x.\,e_4) : \llbracket\tau_1 \to \tau_2\rrbracket_\diamond; \llbracket\sigma_1 \to \sigma_2\rrbracket_\diamond$	By compatibility of coercions
$\diamond \vdash \tau_1 <: \sigma_1$	By inversion

$\diamond \vdash \sigma_2 <: \tau_1$	By inversion
$\diamond \vdash \sigma_2 <: \sigma$	By rule FSUB-TRANS
$\bullet \vdash [\![\tau_1]\!]_\diamond <: [\![\sigma_1]\!]_\diamond \rightsquigarrow co_2$	
$\bullet \vdash [\![\sigma_2]\!]_\diamond <: [\![\sigma]\!]_\diamond \rightsquigarrow co_4$	
$\diamond \vdash \upsilon : \sigma_1 \rightsquigarrow E_3 : [\![\sigma_1]\!]_\diamond$	By rule F-SUB
$\bullet;\bullet \vdash E_3 : [\![\sigma_1]\!]_\diamond \Rightarrow [\![\sigma_1]\!]_\diamond \rightsquigarrow co_2\ e_3$	By rule T-SUB
$\bullet;\bullet \vdash e_3 \backsimeq_{log} co_2\ e_3 : [\![\tau_1]\!]_\diamond; [\![\sigma_1]\!]_\diamond$	By compatibility of coercions
$\diamond \vdash [\upsilon/x]\epsilon : \sigma_2 \rightsquigarrow [(E_3 : [\![\sigma_1]\!]_\diamond)/x]E_4$	By Lemma 22
$\diamond \vdash [\upsilon/x]\epsilon : \sigma \rightsquigarrow [(E_3 : [\![\sigma_1]\!]_\diamond)/x]E_4 : [\![\sigma]\!]_\diamond$	By rule F-SUB
$\bullet;\bullet \vdash [(E_3 : [\![\sigma_1]\!]_\diamond)/x]E_4 : [\![\sigma]\!]_\diamond \Rightarrow [\![\sigma]\!]_\diamond \rightsquigarrow co\,([co_2\ e_3/x]e_4)$	By substitution lemma of F_i^+
$\bullet;\bullet \vdash (\lambda x.\ e_4)\,(co_2\ e_3) \backsimeq_{log} [co_2\ e_3/x]e_4 : [\![\sigma_2]\!]_\diamond$	Reduction preserves logical equivalence
$\bullet;\bullet \vdash (\lambda x.\ e_4)\,(co_2\ e_3) \backsimeq_{log} co_4\,([co_2\ e_3/x]e_4) : [\![\sigma_2]\!]_\diamond; [\![\sigma]\!]_\diamond$	By compatibility of coercions
$\bullet;\bullet \vdash e_2\ e_3 \backsimeq_{log} (\lambda x.\ e_4)\,(co_2\ e_3) : [\![\tau_2]\!]_\diamond; [\![\sigma_2]\!]_\diamond$	By compatibility of applications
$\bullet;\bullet \vdash e_1 \backsimeq_{log} co_4\,([co_2\ e_3/x]e_4) : [\![\sigma]\!]_\diamond$	By transitivity

- The case for rule F-RED-TAPP is similar to rule F-RED-APP, with resort to Lemma 23.
- Case

F-RED-PROJ

$$\frac{}{\{l_1 = \upsilon_1, .., l_i = \upsilon_i, .., l_n = \upsilon_n\}.l \longrightarrow \upsilon_i}$$

$\diamond \vdash \{l_1 = \upsilon_1, .., l_i = \upsilon_i, .., l_n = \upsilon_n\}.l : \sigma \rightsquigarrow E_1$	Given
$\bullet;\bullet \vdash E_1 \Rightarrow [\![\sigma]\!]_\diamond \rightsquigarrow e_1$	Given
$\diamond \vdash \{l_1 = \upsilon_1, .., l_i = \upsilon_i, .., l_n = \upsilon_n\}.l_i : \sigma_i \rightsquigarrow$ $((\{l_1 = E_1'\},, .., , \{l_i = E_i\},, .., , \{l_n = E_n\}) : \{l_i : [\![\sigma_i]\!]_\diamond\}).l$	By inversion
$\diamond \vdash \sigma_i <: \sigma$	
$\bullet \vdash [\![\sigma_i]\!]_\diamond <: [\![\sigma]\!]_\diamond \rightsquigarrow co$	
$\bullet;\bullet \vdash ((\{l_1 = E_1'\},, .., , \{l_i = E_i\},, .., , \{l_n = E_n\}) : \{l_i : [\![\sigma_i]\!]_\diamond\}).l \Rightarrow$ $[\![\sigma_i]\!]_\diamond \rightsquigarrow co_1\ (e_1', .., e_i, .., e_n)$	
$\bullet \vdash \{l_i : [\![\sigma]\!]_\diamond\} <: \{l_1 : [\![\sigma_1]\!]_\diamond, .., l_i : [\![\sigma_i]\!]_\diamond, .., l_n : [\![\sigma_n]\!]_\diamond\} \rightsquigarrow co_1$	
$\bullet;\bullet \vdash e \backsimeq_{log} co\ (co_1\ (e_1', .., e_i, .., e_n)) : [\![\sigma]\!]_\diamond$	By coherence
$\diamond \vdash \upsilon_i : \sigma \rightsquigarrow E_i : [\![\sigma]\!]_\diamond$	By inversion
$\bullet;\bullet \vdash E_i : [\![\sigma]\!]_\diamond \Rightarrow [\![\sigma]\!]_\diamond \rightsquigarrow co\ e_i$	
$co_1\ (e_1', .., e_i, .., e_n) \longrightarrow^* \upsilon_i$	By reduction
$e_i \longrightarrow^* \upsilon_i$	By reduction
$\bullet;\bullet \vdash co_1\ (e_1', .., e_i, .., e_n) \backsimeq_{log} e_i : [\![\sigma_i]\!]_\diamond$	forward an backward reduction preserves equivalence
$\bullet;\bullet \vdash co\ (co_1\ (e_1', .., e_i, .., e_n)) \backsimeq_{log} co\ e_i : [\![\sigma]\!]_\diamond$	
$\bullet;\bullet \vdash e \backsimeq_{log} co\ e_i : [\![\sigma]\!]_\diamond$	By transitivity

◀

▶ **Theorem 21** (Simulation). *If* $\epsilon_1 \longrightarrow \epsilon_2$*, and* $\diamond \vdash \epsilon_2 : \sigma \rightsquigarrow E_2$*, and* $\bullet;\bullet \vdash E_2 \Rightarrow [\![\sigma]\!]_\diamond \rightsquigarrow e_2$*, then there exist* E_1*,* e_1*,* e_1' *such that* $\diamond \vdash \epsilon_1 : \sigma \rightsquigarrow E_1$*, and* $\bullet;\bullet \vdash E_1 \Rightarrow [\![\sigma]\!]_\diamond \rightsquigarrow e_1$*, and* $e_1 \longrightarrow e_1'$*, where* $\bullet;\bullet \vdash e_1' \backsimeq_{ctx} e_2$*.*

Proof. Follows directly from Lemma 24, the theorem that operational semantics preserves logical equivalence, and the theorem that logical equivalence implies contextual equivalence.

◀