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 $F_{<}$ 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.

2012 ACM Subject Classification

Theory of computation → Type theory; Software and its engineering → Object oriented languages; Software and its engineering → Polymorphism

Keywords and phrases
Intersection types, bounded polymorphism, row polymorphism

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.27

Supplementary Material https://github.com/xnning/Row-and-Bounded-via-Disjoint

Funding This work has been sponsored by Hong Kong Research Grant Council projects number 17210617 and 17209519, and by the Research Foundation - Flanders.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Intersection types [51, 22, 59] 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 [52] 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
Row and Bounded Polymorphism via Disjoint Polymorphism

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]:

```javascript
```

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:

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

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 can type-check flexible OO patterns that have been used for many years in many dynamically-typed languages. Functions similar to `extend` have also been encoded in Scala [47, 54].

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 [55]. As illustrated by Dunfield [28], the expressive power of the merge operator can encode 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:

```haskell
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 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 with disjoint polymorphism. Indeed, there are clear analogies between the two mechanisms: the merge operators (,, and ||) are 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:

```plaintext
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:

```plaintext
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:

```plaintext
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 [48].

Disjoint polymorphism, row polymorphism and bounded quantification provide a range of functionalities for OOP languages. Thus 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. Yet 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. Theoretically, it is important to formally establish the comparison among different type features, to allow a deep understanding and a precise discussion of the relative expressiveness of each feature. In practice, this result suggests
that core languages wishing to support all those features only need to support disjoint polymorphism natively, promising an economy of the 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 $F_<$ [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 $F_<$ (the most widely used and decidable fragment of $F_<$) 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 $F_<$ that is encodable in terms of polymorphic intersection type systems, by providing an elaboration from kernel $F_<$ 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 $F_<$ 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 $F_<$ 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 $F_i^+$).

## 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 $1,,\text{True}$ of type $\text{Int} \& \text{Bool}$. Thanks to subtyping, a term of type $\text{Int} \& \text{Bool}$ can also be used as if it had type $\text{Int}$, or as if it had type $\text{Bool}$. $F_i^+$ requires the two components of a merge to
have disjoint types, e.g., \(1, 2 : \text{Int} \& \text{Int}\) is not allowed, because it is ambiguous which value should be used at type \(\text{Int}\). With disjoint quantification, it is possible to merge components whose type contains type variables. For instance, the term \(\Lambda(\alpha \to \text{Int}). \lambda(x : \alpha). x, 1\) has type \(\forall(\alpha \to \text{Int}). \alpha \to \alpha \& \text{Int}\). The disjointness annotation \(\alpha \to \text{Int}\) allows \(\alpha\) to be instantiated only to types that are disjoint from \(\text{Int}\). Without a disjointness constraint, the term \(\Lambda\alpha. \lambda(x : \alpha). x, 1\) is rejected. Otherwise such a term would allow \(\alpha\) to be instantiated to \(\text{Int}\), and thus the function could be applied to numbers, e.g., 2, leading to the ambiguous merge \(2, 1\).

2.2 Row Polymorphism through Disjoint Polymorphism

Row types, originally introduced by Wand [63] to model inheritance, provide an approach to typing extensible records. Row types have been studied extensively [35, 11, 53, 42] and have been applied to provide extensibility in various type systems [37, 36, 38]. According to Rémy [53], record calculi can be divided into those that support strict 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 \(\lambda^\parallel\), a calculus proposed by Harper and Pierce [34] that extends System F with row polymorphism. \(\lambda^\parallel\) belongs to the strict 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 \(\lambda^\parallel\) 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 \(||\):

\[
\text{mergeRcd} = \Lambda(\alpha_1 \# \text{Empty}). \Lambda(\alpha_2 \# \alpha_1). \lambda(x_1 : \alpha_1). \lambda(x_2 : \alpha_2). x_1 || x_2
\]

which takes two type variables, each of which lacks \((\#)\) the appropriate fields (\text{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 \((\alpha_2 \neq \alpha_1)\), i.e., the second record cannot have any labels that also occur in the first. These constraints ensure that the resulting record \(x_1 || x_2\) 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 \(\text{Int}\), we can refine the constraint list of \(\alpha_1, \alpha_2\) and the type of \(x_1\) accordingly:

\[
\Lambda(\alpha_1 \# \{l_1 : \text{Int}\}). \Lambda(\alpha_2 \# (\alpha_1, \{l_1 : \text{Int}\})). \lambda(x_1 : \alpha_1 || \{l_1 : \text{Int}\}). \lambda(x_2 : \alpha_2). x_1 || x_2
\]

Encoding with disjoint polymorphism Our encoding of \(\lambda^\parallel\) into \(F^+_1\) 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 \(x_1 || x_2\) and the merge operator \(x_1, x_2\). Exploiting these similarities, we can encode \(\text{mergeRcd}\) as follows in \(F^+_1\):

\[
\text{mergeAny} = \Lambda(\alpha_1 \# T). \Lambda(\alpha_2 \# \alpha_1). \lambda(x_1 : \alpha_1). \lambda(x_2 : \alpha_2). x_1, x_2
\]

An important difference is that in \(\text{mergeRcd}\), \(\alpha_1\) and \(\alpha_2\) are row variables: they can only be instantiated with record types. In contrast in \(\text{mergeAny}\), \(\alpha_1\) and \(\alpha_2\) are type variables and they can be instantiated with any types, including types which are not records (such as \(\text{Int}\)).

Formal elaboration To establish the validity of the encoding, we have formalized two different elaborations of \(\lambda^\parallel\) into \(F^+_1\). 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 $\Lambda \alpha \# \{ l : \text{Int} \}$ expresses that $\alpha$ cannot have the label $l$ at all. In contrast, the $F^+$ binder $\Lambda \beta * \{ l : \text{Int} \}$ expresses that $\beta$ cannot have a field $l$ of type $\text{Int}$, but it can have a field $l$ of some other disjoint type, say $\text{Bool}$. 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, 48, 49, 13, 11, 18, 25, 50]. In this paper, we focus on System $F^<$, which is a calculus with bounded quantification that extends System $F$.

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

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

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

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

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

**Encoding with disjoint polymorphism** Pierce [48] informally discussed an encoding of bounded quantification in terms of intersection types. To illustrate the encoding, let us consider a function of type $\forall (\alpha <: \text{Int}). \alpha \to \alpha$, which requires the type of the argument to be a subtype of $\text{Int}$. With intersection types, we know that $\alpha \& \text{Int}$ is always a subtype of $\text{Int}$. Therefore, the type $\forall \alpha. (\alpha \& \text{Int}) \to (\alpha \& \text{Int})$ expresses a similar subtype requirement. This leads to the following encoding of bounded quantification, 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 $fpoly$ example, we have its encoded type

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

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 $F^<$, 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 $F_{<}$, i.e., kernel $F_{<}$, 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 $F_{<}$ and $F_{+}$ 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 $F_{<}$ 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 $F_{+}$ 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 $F_{<}$ 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 $F_{<}$ programs are encodable as well-typed $F_{+}$ programs. We believe that this justifies Pierce’s claim that the encoding might work better than expected, as kernel $F_{<}$ is the most common decidable fragment of $F_{<}$ 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 $F_{+}$ over $F_{<}$, and $\lambda$ alone.

**Distributivity, Nested Composition and Family Polymorphism**  $F_{+}$ 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:

```plaintext
type R[e] = {lit : Int → e, neg : e → e} -- literal and negative expressions
compose = \( a * \top \). \( b * a \). \( r1 : R[a] \). \( r2 : R[b] \). (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 $F_{+}$ 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 $F_{+}$, 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) [62]. 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:

\[
\Lambda(a \# \text{Empty}). \Lambda(b \# a). \lambda(r1 : R[a]). \lambda(r2 : R[b]). \\
\begin{cases}
\text{lit} = \lambda(i : \text{Int}) . (r1.\text{lit } i, r2.\text{lit } i) \\
\text{neg} = \lambda(e : (a, b)) . (r1.\text{neg } (\text{fst } e), r2.\text{neg } (\text{snd } e))
\end{cases}
\]

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 [65]. 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** \( F^+ \) 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:

\[
\lambda(x : \{l : \text{Int}\}). x
\]

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

\( \lambda\|\) solves the loss of information problem by formulating the function in a different way:

\[
\Lambda(a \# \{l : \text{Int}\}). \lambda(x : \{l : \text{Int}\} \| a). x
\]

In this function the row variable \( a \) 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 \( a \). In this function the output type is \( \{l : \text{Int}\} \| a \) as well. Therefore the application of the function to \( \{l = 1, l' = \text{True}\} \) has the type \( \{l : \text{Int}, l' : \text{Bool}\} \), which does not lose precision.

In \( F^+ \) we can easily translate the \( \lambda\| \) approach and reap its benefits too:

\[
\Lambda(\alpha \ast \{l : \text{Int}\}). \lambda(x : \{l : \text{Int}\} \& a). x
\]

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:

\[
\lambda(x : \{l : \text{Int}\}). x.l + 1
\]

The function has type \( \{l : \text{Int}\} \rightarrow \text{Int} \). 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 \( F^+ \) and, because of subtyping, the record \( \{l = 1, l' = \text{True}\} \) is a valid argument. However in \( \lambda\| \), the only way to allow records with more labels, is to generalize the function to:

\[
\Lambda(\alpha \# \{l : \text{Int}\}). \lambda(x : \{l : \text{Int}\} \| a). x.l + 1
\]

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 prior work on 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 $\text{Int}$, the top type $\top$, the bottom type $\bot$, arrows $A \to B$, intersection types $A \& B$, singleton record types $\{l : A\}$, type variables $\alpha$ and disjoint quantification $\forall \alpha \cdot A$. Expressions $E$ include term variables $x$, integers $i$, the top value $\top$, abstractions $\lambda x \cdot E$, applications $E_1 E_2$, merge expressions $E_1 \,\ll\, E_2$, annotated terms $E : A$, singleton records $\{l = E\}$, record projections $E.l$, type abstractions $\Lambda(\alpha \cdot A)$. $E$ and type applications $E A$. Term contexts $\Gamma$ record types of term variables, and type contexts $\Delta$ 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.

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 $\Delta$ and the term context $\Gamma$, 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 $\Delta$ and $\Gamma$. Most of the typing rules are standard. Rule T-merge says that the merge expression $E_1 \,\ll\, E_2$ is well-typed if both sub-expressions are well-typed, and their types are disjoint. The disjointness judgment $\Delta \vdash A_1 \ast A_2$ is important to rule out invalid merges, such as $1 \,\ll\, 2$. Rule T-tapp 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 $\uparrow A$, given at the top of Figure 2, captures the set of types that are isomorphic to $\top$. Disjointness axioms $A \ast \alpha B$ (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.
### Types

\[ A, B, C :\equiv \text{Int} | \top | \bot | A \rightarrow B | A \& B | \{1 : A\} | \alpha | \forall (\alpha \ast A). B \]

### Expressions

\[ E :\equiv x \mid i \mid \top \mid \lambda x. E \mid E_1 E_2 \mid E_1 \& E_2 \mid E : A \mid \{1 = E\} | E.1 \]

### Term contexts

\[ \Gamma :\equiv \bullet | \Gamma, x : A \]

### Type contexts

\[ \Delta :\equiv \bullet | \Delta, \alpha \ast A \]

<table>
<thead>
<tr>
<th>A &lt;: B</th>
<th>(Declarative subtyping)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-refl</td>
<td>S-trans</td>
</tr>
<tr>
<td>S-top</td>
<td>S-bot</td>
</tr>
<tr>
<td>S-distRcd</td>
<td>A &lt;: B</td>
</tr>
</tbody>
</table>

| S-arr | S-forall | S-arr | S-andl |

| S-andRcd | S-distAll |
| (A1 \rightarrow A2) \& (A1 \rightarrow A3) <: A1 \rightarrow A2 & A3 |

| \{1 : A\} \& \{1 : A\} <: \{1 : A \& B\} | \{1 : A \& B\} <: \forall (\alpha \ast A). B1 & B2 |
| S-topRcd | S-topRcd |
| \top <: \{1 : \top\} | \top <: \forall (\alpha \ast \top). \top |

<table>
<thead>
<tr>
<th>\Delta, \Gamma \vdash E \Rightarrow A</th>
<th>(Inference)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-top</td>
<td>Δ, Γ \vdash E \Rightarrow A</td>
</tr>
<tr>
<td>Δ, Γ \vdash T \Rightarrow Γ</td>
<td></td>
</tr>
<tr>
<td>Δ, Γ \vdash \top \Rightarrow \top</td>
<td></td>
</tr>
<tr>
<td>Δ, Γ \vdash i \Rightarrow \text{Int}</td>
<td></td>
</tr>
</tbody>
</table>

| T-nat | Δ, Γ \vdash \top |
| Δ, Γ \vdash \alpha \Rightarrow \text{Int} |
| Δ, Γ \vdash (x : A) \in \Gamma |

| T-app | Δ, Γ \vdash E_1 \Rightarrow A_1 |
| Δ, Γ \vdash E_2 \Rightarrow A_2 |
| Δ, Γ \vdash E_1 E_2 \Rightarrow A_2 |

| T-merge | Δ, Γ \vdash \forall (\alpha \ast A). E \Rightarrow \forall (\alpha \ast A). B |
| Δ, Γ \vdash A \Rightarrow \forall (\alpha \ast A). B |
| Δ, Γ \vdash A \Rightarrow \forall (\alpha \ast A). B |

| Ttabs | Δ, Γ \vdash \forall (\alpha \ast A). B \Rightarrow \forall (\alpha \ast A). E |
| Δ, Γ \vdash \forall (\alpha \ast A). C \Rightarrow \forall (\alpha \ast A). B |

| T-annino | Δ, Γ \vdash E_1 \Rightarrow A_1 |
| Δ, Γ \vdash E_2 \Rightarrow A_2 |
| Δ, Γ \vdash E_1 \& E_2 \Rightarrow \alpha \ast A_2 |

| T-project | Δ, Γ \vdash \forall (\alpha \ast A). B \Rightarrow \forall (\alpha \ast A). C |
| Δ, Γ \vdash \forall (\alpha \ast A). C \Rightarrow \forall (\alpha \ast A). B |
| Δ, Γ \vdash \forall (\alpha \ast A). C \Rightarrow \forall (\alpha \ast A). B |

| T-apply | Δ, Γ \vdash E \Rightarrow \alpha \ast A_2 |
| Δ, Γ \vdash E \Rightarrow \alpha \ast A_2 |
| Δ, Γ \vdash E \Rightarrow \alpha \ast A_2 |

| T-tabs | Δ, Γ \vdash A \Rightarrow \alpha \ast A_2 |
| Δ, Γ \vdash A \Rightarrow \alpha \ast A_2 |
| Δ, Γ \vdash A \Rightarrow \alpha \ast A_2 |

| T-sub | Δ, Γ \vdash \alpha \ast A_2 |
| Δ, Γ \vdash \alpha \ast A_2 |
| Δ, Γ \vdash \alpha \ast A_2 |

| T-checking | Δ, Γ \vdash E \Rightarrow A |
| Δ, Γ \vdash E \Rightarrow A |
| Δ, Γ \vdash E \Rightarrow A |

| Figure 1 | Syntax, declarative subtyping, and bidirectional type system of \( F^+ \). |
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 $\lambda^\parallel$, and then present a carefully crafted elaboration of full $\lambda^\parallel$ using a more sophisticated elaboration.

### 4.1 Syntax of $\lambda^\parallel$

We start by briefly reviewing the syntax of $\lambda^\parallel$, shown at the top of Figure 3. Metavariable $t$ ranges over types, which include the integer type $\text{Int}$, function types $t_1 \to t_2$, constrained quantifications $\forall \alpha \neq R.t$ and record types $r$. Record types are built from record type variables $\alpha$, the empty record type $\text{Empty}$, single-field records $\{l : t\}$ and record merges $r_1 \parallel r_2$.\(^1\) A constraint list $R$ of record types is used to constrain instantiations of record type variables.

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

### 4.2 Typing Rules of $\lambda^\parallel$

The type system of $\lambda^\parallel$ 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 $r_1 \parallel r_2$ 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 $\text{wfr-Merge}$)—the most important judgment in $\lambda^\parallel$, as we will explain next.

**Compatibility** The compatibility relation in the middle of Figure 3 plays a central role in $\lambda^\parallel$. It is the underlying mechanism for deciding when merging two records is “sensible”. Informally, $T \vDash r_1 \neq r_2$ holds if $r_1$ lacks every field contained in $r_2$ and vice versa. Compatibility is symmetric (rule $\text{cmp-Symm}$) and respects type equivalence (rule $\text{cmp-Eq}$). Rule $\text{cmp-Base}$ 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 $\text{cmp-Tvar}$). Two single-field records are compatible if they have different labels (rule $\text{cmp-BaseBase}$). The remaining rules are self-explanatory: we refer the reader to [34] for further explanation. The judgment of constraint list satisfaction $T \vDash r \neq R$ ensures that $r$ is compatible with every record in the constraint list $R$.

**Type equivalence** Unlike $\lambda^\parallel$, $\lambda^\parallel$ does not have subtyping. Instead, $\lambda^\parallel$ 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 $\text{teq-MergeAssoc}$), commutative (rule $\text{teq-MergeComm}$), and has $\text{Empty}$ as its unit (rule $\text{teq-MergeUnit}$). As a consequence, records are identified up to permutations. The equivalence of constrained quantification (rule $\text{teq-CongAll}$) 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 $\text{ceq-Swap}$). Merges of constraints can be “flattened” (rule $\text{ceq-Merge}$), and occurrences of $\text{Empty}$ may be

\(^1\) The original $\lambda^\parallel$ also includes record type restrictions $r \backslash l$, which can be systematically erased using type equivalence, thus we omit type-level restrictions but keep term-level restrictions.
Types

\[ t ::= \text{Int} \mid t_1 \rightarrow t_2 \mid \forall \alpha \# R.t \mid r \]

Records

\[ r ::= \alpha \mid \text{Empty} \mid \{ l : t \} \mid r_1 || r_2 \]

Constraint lists

\[ R ::= \text{Empty} \mid r \mid R \]

Terms

\[ \varepsilon ::= x \mid i \mid \lambda(x : t).\varepsilon \mid \varepsilon_1 \varepsilon_2 \mid \text{empty} \mid \{ l = \varepsilon \} \mid \varepsilon_1 || \varepsilon_2 \]

Term contexts

\[ G ::= \text{Empty} \mid G, x : t \]

Type contexts

\[ T ::= \text{Empty} \mid T, \alpha \# R \]

---

**Figure 3** Syntax, and selected rules of \( \Lambda^l \).
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).

\begin{itemize}
\item \textbf{Remark 3.} We have made a few simplifications compared to the original $\lambda^\parallel$, 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].
\end{itemize}

\subsection{4.3 A Simple yet Incomplete Encoding}

The similarities between $\lambda^\parallel$ and $\mathbf{F}_{i}^+$, 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 $\lambda^\parallel$ types and $\mathbf{F}_{i}^+$ types. We use $[t]$ to denote the elaboration
function from $\lambda^\parallel$ types to $\mathbf{F}_{i}^+$ types, whose formal definition is given at the top of Figure 5.
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
Allowing it to type-check causes incoherence: evaluating $\lambda x.\, \Lambda(\alpha \# R).\, \exists\forall\alpha \# R.\, t \sim r \# R$.

**Example 4.** Consider the term $\Lambda(\alpha \# \{\, : \text{Int}\}).\, \lambda(x : \alpha).\, x$. This term can be assigned the type (among others) $\forall\alpha \# \{\, : \text{Int}\}.\, \alpha \rightarrow \alpha$, and its $F^+_1$ counterpart $\Lambda(\alpha \# \{\, : \text{Int}\}).\, \lambda(x : \alpha).\, x$ has type $\forall(\alpha \# \{\, : \text{Int}\}).\, \alpha \rightarrow \alpha$, which corresponds directly to $\forall\alpha \# \{\, : \text{Int}\}.\, \alpha \rightarrow \alpha$. In $\lambda^\|$, the same term could also be assigned type $\forall\alpha \# \{\, : \text{Bool}\}.\, \alpha \rightarrow \alpha$ (rule wtti-Eq), since $\forall\alpha \# \{\, : \text{Bool}\}.\, \alpha \rightarrow \alpha$ is equivalent to $\forall\alpha \# \{\, : \text{Int}\}.\, \alpha \rightarrow \alpha$ by rules teq-CongAll and ceq-Base. However, in $F^+_1$, these two types have no relationship at all—$\forall(\alpha \# \{\, : \text{Int}\}).\, \alpha \rightarrow \alpha$ is not the same as $\forall(\alpha \# \{\, : \text{Bool}\}).\, \alpha \rightarrow \alpha$, and indeed it should not be, as these two types have completely different meanings!

**Example 5.** Consider the term $\varepsilon = \Lambda(\alpha \# \{\, : \text{Bool}\}).\, \lambda(x : \alpha).\, \lambda(y : \{\, : \text{Int}\}).\, x \parallel y$. This term has type $\forall\alpha \# \{\, : \text{Bool}\}.\, \alpha \rightarrow \{\, : \text{Int}\} \rightarrow \alpha \parallel \{\, : \text{Int}\}$, and its “obvious” elaboration is $E = \Lambda(\alpha \# \{\, : \text{Bool}\}).\, \lambda(x : \alpha).\, \lambda(y : \{\, : \text{Int}\}).\, x \parallel y$. However, expression $E$ is ill-typed in $F^+_1$: we cannot merge $x$ with $y$ because their types ($\alpha$ and $\{\, : \text{Int}\}$ respectively) are not disjoint. Allowing it to type-check causes incoherence: evaluating $(E\, \{\, : \text{Int}\} \{1 = l\} \{1 = l\} \{2\})\.l$ could result in 1 or 2!

These examples underline a crucial observation: disjointness is more fine-grained than compatibility. Unlike $F^+_1$, the existence of $\varepsilon$ in $\lambda^\|$ will not cause incoherence because compatibility can only relate records with different labels, and thus $\varepsilon$ can only be applied to
records without label \( l \) at all. So \( \lambda \) rejects type application \( \varepsilon \cdot \{ l : \text{Int} \} \) 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\(_1\). 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.

\[
\begin{array}{c}
\text{CEQ-Base} \\
\{ l : t \}, R \sim \{ l : t' \}, R \\
T \vdash x \# \{ l : t \} \\
T \vdash t' \text{ type} \\
\text{CMP-Base} \\
T \vdash x \# \{ l : t' \}
\end{array}
\]

Yet, both Example 4 and Example 5 seem contrived. From the expression \( \Lambda(\alpha \# \{ l : \text{Int} \}) \cdot \lambda(x : \alpha).x, \) the user can reasonably expect the type to be \( \forall \alpha \# \{ l : \text{Int} \}. \alpha \rightarrow \alpha \). For \( \varepsilon \), an equivalent definition with more sensible and readable annotation is \( \varepsilon' = \Lambda(\alpha \# \{ l : \text{Int} \}) \cdot \lambda(x : \alpha). \lambda(y : \{ l : \text{Int} \}). x || y, \) 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 \( \alpha_i \)'s are record type variables. The normalization then validates the change of rules.

\[
\begin{array}{c}
\text{CEQ-BaseAlt} \\
\{ l : t \}, R \sim \{ l : t' \}, R \\
t \sim t' \\
\end{array}
\]

In return, we can prove the intuitive elaboration for restricted \( \lambda \) is, indeed, sound:

\textbf{Theorem 6 (Type-safety of \( \sim \), elaboration). If } T; G \vdash \varepsilon : t \sim \varepsilon, E \text{ 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 \( F^\dagger \) 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 \( F^\dagger \), we utilize the bottom type \( \bot \). In \( F^\dagger \), 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 \( F^\dagger \), if a record is disjoint to \( \{ l : \bot \} \), then it is disjoint to every single-field record \( \{ l : A \} \).

\textbf{Lemma 7 (Disjointness to records with bottom). If } \Delta \vdash A \ast \{ l : \bot \}, \text{ then } \Delta \vdash A \ast \{ l : B \} \text{ for all } B.\]

Essentially, a compatibility constraint with \( \{ l : t \} \) in \( \lambda \) corresponds to a disjointness constraint to \( \{ l : \bot \} \) in \( F^\dagger \). 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 \( \forall \alpha \# \{ l : \text{Int} \}. \alpha \rightarrow \alpha \) and \( \forall \alpha \# \{ l : \text{Bool} \}. \alpha \rightarrow \alpha \) elaborate to
∀(α ∗ {l : ⊥}). α → α. For Example 5, ε elaborates to $E' = \Lambda(\alpha ∗ \{l : \bot\}). \lambda(x : \alpha). \lambda(y : \{l : \text{Int}\}). x, y$, which type-checks in $F^+_1$.

Example 8. Now consider the $\lambda^l$ term

$\varepsilon_1 = (\Lambda(\alpha ∗ \top). \lambda(x : (\forall \beta ∗ \alpha. \text{Int})). 1) \{\{l : \text{Int}\}\} (\Lambda(\beta ∗ \{l : \bot\}). 2)$

The term type-checks in $\lambda^l$ and has type $\text{Int}$. During elaboration, we treat records differently according to where they occur. For the type argument $\{l : \text{Int}\}$, since it is not in a constraint list, we elaborate it normally to $\{l : \text{Int}\}$. For the term argument $(\Lambda(\beta ∗ \{l : \bot\}). 2)$, since the record $\{l : \text{Int}\}$ appears in a constraint list, we elaborate the term argument to $(\Lambda(\beta ∗ \{l : \bot\}). 2)$. The whole term is then elaborated to $E_1 = (\Lambda(\alpha ∗ \top). (\lambda(x. 1) : \forall(\beta ∗ \alpha. \text{Int}) → \text{Int})) \{\{l : \text{Int}\}\} (\Lambda(\beta ∗ \{l : \bot\}). 2)$

However, $E_1$ fails to type-check under $F^+_1$: after type application, we substitute $\alpha$ with the type argument $\{l : \text{Int}\}$ in $x$’s type $\forall(\beta ∗ \alpha. \text{Int})$, yielding $\forall(\beta ∗ \{l : \bot\}). \text{Int}$, whereas the term argument has type $\forall(\beta ∗ \{l : \bot\}). \text{Int}$, which does not match (and is not a subtype of) the expected parameter type $\text{Int}$.

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., $(\forall(\beta ∗ \{l : \bot\}). \text{Int}) → \text{Int}$, has $\{l : \text{Int}\}$ 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, $(\forall(\beta ∗ \{l : \bot\}). \text{Int}) → \text{Int})$, $\{l : \text{Int}\}$ should not elaborate to $(\forall(\beta ∗ \top). (\lambda(x. x : \alpha → \alpha)) \{l : \bot\} \{l = 1\}). l + 1$, 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 $\alpha$ to a pair of type variables $\alpha$ 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 $\alpha_1$ or $\alpha_2$ 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 $\alpha$ takes the normal type argument, $\alpha_\bot$ takes an extra bottom-elaborated type argument. As an example, the expression $\varepsilon_1$ in Example 8 is elaborated to $E'_1$, which type-checks successfully in $F^+_1$, where the differences from $E_1$ are highlighted in gray.

$E'_1 = (\Lambda(\alpha ∗ \top). \lambda(x. 1) : \forall(\beta ∗ \alpha_\bot). \text{Int}) → \text{Int}) \{\{l : \text{Int}\}\} (\Lambda(\beta ∗ \{l : \bot\}). 2)$

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

Example 9. Consider the term

$\varepsilon_2 = \Lambda(\alpha ∗ \top). \lambda(x : \alpha_\bot). \lambda(y : \beta). x \parallel y$.

Under the current approach, it elaborates to $E_2 = \Lambda(\alpha ∗ \top). \lambda(x : \alpha_\bot). \lambda(y : \beta). x, y$.

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 $F^+_1$ 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 regularly elaborated types as well as the bottom-elaborated types into the disjointness
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 \([r]_\perp\) for bottom elaboration. In regular elaboration \([t]\), \(\alpha\) elaborates to \(\alpha\). Of particular interest is the case of elaborating quantifiers: each quantifier \(\forall \alpha \neq R, t\) is split into two quantifiers \(\forall (\alpha \ast [R])\). \(\forall (\alpha \ast [R])\). \([t]\) in \(F^+_\perp\). The relative order of \(\alpha\) and \(\alpha \perp\) is not important, as long as we respect the order when elaborating type applications. Bottom elaboration \([r]_\perp\) elaborates \(\alpha\) to \(\alpha \perp\), 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 \(\beta\) is compatible with \(\alpha\), then its elaboration \(\beta\) is disjoint with both \(\alpha\) and \(\alpha \perp\).

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

4.6 Metatheory

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

\[\text{Lemma 10 (Elaboration commutes with substitution). We have (1) } [r/\alpha]t = [r/\alpha_\perp][r/\alpha][t];\]
\[\text{and (2) } [r/\alpha]\alpha_\perp = [r/\alpha_\perp][r/\alpha][\alpha_\perp]; \text{ and (3) } [r/\alpha]R = [r/\alpha_\perp][r/\alpha][R].\]
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 \([t_1] \prec [t_2]\) and \([t_2] \prec [t_1]\).

- **Lemma 12** (Compatibility implies disjointness). If \( T \vdash r \# r_2 \), then we have: (1) \([T] \vdash [r_1] \times [r_2];\) (2) \([T] \vdash [r_1] \times \bot;\) (3) \([T] \vdash [r_1] \times [r_2];\) and (4) \([T] \vdash [r_1] \times [r_2]\).

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

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 \( T; G \vdash e : t \Rightarrow E \), then \([T]; [G] \vdash E \Rightarrow [t] \).

**Coherence**
Because of rule wtt-Eq, a \( \lambda^l \) expression can possibly elaborate to many different \( F^+ \) expressions. For example, the term \( \Lambda(\alpha \# \{l : \text{Int}\}). \lambda(x : \alpha). x \) has the following two elaborations \( E_1 \) and \( E_2 \) (among others). This is the problem of coherence [56]: the meaning of a target program depends on the choice of a particular elaboration typing.

1. \( E_1 = \Lambda(\alpha \# \{l : \text{Int}\}) \cdot \Lambda(\alpha) \cdot \lambda(x : \alpha). x; \)
2. \( E_2 = (E_1 : [\forall \alpha \# \{l : \text{Int}\}, \alpha \rightarrow \alpha]) \cdot [\forall \# \{l : \text{Int}\}, \alpha \rightarrow \alpha] \)

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

- **Theorem 15** (Coherence of elaboration). If \( \circ; \circ \vdash e : t \Rightarrow E_1, \) and \( \circ; \circ \vdash \circ : t \Rightarrow E_2, \) and \( \bullet; \bullet \vdash E_1 \Rightarrow [t] \Rightarrow e_1, \) and \( \bullet; \bullet \vdash E_2 \Rightarrow [t] \Rightarrow e_2, \) then \( \circ; \circ \vdash e_1 \subseteq_{ctx} e_2. \)

5 Encoding Bounded Quantification

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

5.1 Syntax and Semantics of kernel \( F_{<} \)

We start by reviewing the syntax and semantics of kernel \( F_{<}, \) a polymorphic calculus with bounded quantification. The syntax of \( F_{<} \) is given at the top of Figure 7. It is a version of \( F_{<} \) extended with records [10]. In addition to standard System F constructs, types \( \sigma \) include bounded quantifications \( \forall(\alpha < \tau) . \sigma, \) which give a bound for the type variable; and record types \( \{l_k : \sigma_1, \ldots, l_n : \sigma_n\}, \) for which we assume all labels are distinct. In addition to standard System F terms, terms \( e \) include type abstractions \( \Lambda(\alpha < \sigma) . e, \) records \( \{l_1 = e_1, \ldots, l_n = e_n\}, \) and projections e.t. Contexts \( \Sigma \) 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.

---

2 We could also encode record types in \( F_{<} \), which however is a bit involved.
| Types | $\sigma, \tau \equiv \text{Int} \mid T \mid \alpha \mid \sigma \to \tau \mid \forall(\alpha < \tau). \sigma \mid \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} |
| Terms | $\epsilon \equiv i \mid T \mid x \mid (x : \sigma). \epsilon \mid \epsilon_1 \epsilon_2 \mid \Lambda(\alpha < \tau). \epsilon \mid \epsilon \mid \epsilon_1 = \epsilon_1, \ldots, l_n = \epsilon_n \mid \epsilon.l |
| Value | $\nu \equiv i \mid T \mid \lambda(x : \sigma). \epsilon \mid \Lambda(\alpha < \sigma). \epsilon \mid \{l_1 = v_1, \ldots, l_n = v_n\} |
| Context | $\Sigma \equiv o \mid \Sigma, x : \sigma \mid \Sigma, \alpha < \sigma |

### Subtyping

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma \vdash \sigma &lt; : \tau$</td>
<td>(Subtyping)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma \vdash \sigma &lt; : \tau$</td>
<td>$\Sigma \vdash \sigma_1 &lt; : \sigma_2$</td>
</tr>
<tr>
<td>$\Sigma \vdash \sigma_2 &lt; : \sigma_3$</td>
<td>$\Sigma \vdash \sigma_1 &lt; : \sigma_3$</td>
</tr>
</tbody>
</table>

### Typing

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma \vdash \epsilon : \sigma \equiv \mathcal{E}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma \vdash \Gamma : \top \equiv \top$</td>
<td>$\Sigma \vdash i : \text{Int} \equiv i$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma \vdash \lambda(x : \sigma). \epsilon : \sigma \to \tau \equiv (\lambda x. \epsilon) : ([\sigma]\Sigma \to [\tau]\Sigma)$</td>
<td>$\Sigma \vdash \epsilon : \sigma \equiv \top \equiv \tau : [\tau]\Sigma$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma \vdash \epsilon_1 : \sigma \to \tau \equiv \mathcal{E}_1$</td>
<td>$\Sigma \vdash \epsilon_2 : \sigma \to \tau \equiv \mathcal{E}_2$</td>
</tr>
<tr>
<td>$\Sigma \vdash \epsilon_1 \epsilon_2 : \tau \equiv \mathcal{E}_1 \mathcal{E}_2$</td>
<td>$\Sigma \vdash \Lambda(\alpha &lt; \sigma). \epsilon : \forall(\alpha &lt; \sigma). \tau \equiv \Lambda(\alpha \to \top). \mathcal{E}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma \vdash \epsilon_1 : \sigma_1 \equiv \mathcal{E}_1$</td>
<td>$\Sigma \vdash \epsilon_2 : \sigma_2 \equiv \mathcal{E}_2$</td>
</tr>
<tr>
<td>$\Sigma \vdash \epsilon_1 \epsilon_2 : \tau \equiv \mathcal{E}_1 \mathcal{E}_2$</td>
<td>$\Sigma \vdash \Lambda(\alpha &lt; \sigma). \epsilon : \forall(\alpha &lt; \sigma). \tau \equiv \Lambda(\alpha \to \top). \mathcal{E}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma \vdash (\mathcal{L}_1 : \mathcal{L}_2)$</td>
<td>$\Sigma \vdash \epsilon : \forall(\alpha &lt; \tau_1). \tau_2 \equiv \tau \equiv \mathcal{E}$</td>
</tr>
<tr>
<td>$\Sigma \vdash \mathcal{E} : [\sigma/\alpha] \equiv (E[[\sigma/\alpha]] \equiv E[[\sigma/\alpha]]) \equiv \mathcal{E}$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[[\sigma]\Sigma]$</td>
<td>$\text{Int}\Sigma = \text{Int}$</td>
</tr>
<tr>
<td>$[[\top]\Sigma]$</td>
<td>$\top \equiv \top$</td>
</tr>
<tr>
<td>$[[\sigma \to \tau]\Sigma]$</td>
<td>$[[\sigma]\Sigma \to [[\tau]\Sigma]$</td>
</tr>
<tr>
<td>$[[{l_1 : \sigma_1, \ldots, l_n : \sigma_n}]\Sigma]$</td>
<td>${l_1 : [[\sigma_1]\Sigma], \ldots, {l_n : [[\sigma_n]\Sigma]$</td>
</tr>
<tr>
<td>$[[\alpha]_{(\Sigma, \beta, &lt; \sigma)}]$</td>
<td>$[[\alpha]\Sigma$</td>
</tr>
<tr>
<td>$[[\alpha]_{(\Sigma, \beta, &lt; \sigma)}]$</td>
<td>$[[\alpha]\Sigma$</td>
</tr>
<tr>
<td>$[[\gamma(\alpha &lt; \sigma) \tau]\Sigma]$</td>
<td>$\forall(\alpha : \top). \tau[\Sigma, \alpha &lt; \sigma]$</td>
</tr>
</tbody>
</table>

---

**Figure 7** Syntax, subtyping, typing and elaboration of kernel $F_{\ll}$.  

---
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 \( \alpha \) is a subtype of its bound \( \sigma \). 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 \( F_< \) 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 \( F_< \) 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 \( F_< \) are shown below the subtyping relation. The reader is advised to ignore the gray parts for now. Most rules are straightforward. Unlike \( F_+ \), \( F_< \) 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 [48] 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 \( \alpha \) with \( \alpha \& \sigma \), we collect the bounds \( \alpha <: \sigma \) as we traverse the quantifiers, and only substitute when we encounter a type variable \( \alpha \). This strategy is consistent with elaborating types with free type variables. For example, consider the expression \( \lambda(x : \alpha). x + 1 : \alpha \rightarrow \text{Int} \). This expression type-checks because we have the information \( \alpha <: \text{Int} \) in the context so that we can upcast (by rule F-SUB) the type of \( x \) to \( \text{Int} \) when checking \( x + 1 \). Here it is important 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 \( \alpha \) under the context \( \alpha <: \text{Int} \), it should give us the same result as if elaborating \( \alpha \) inside \( \forall (\alpha <: \text{Int}) \). Therefore, in this case, we substitute \( \alpha \) by \( \alpha \& \text{Int} \), which yields \( x : \alpha \& \text{Int} \), and thus validates \( x + 1 \).

Formally, type elaboration is denoted as \( [\sigma]_{\Sigma} = A \), which reads: under context \( \Sigma \), type \( \sigma \) elaborates to type \( A \). Elaboration of a closed type is just a special case where the context is empty, i.e., \( [\sigma]_{\epsilon} \). 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 \( \alpha \), we traverse the context until we find its subtyping constraint \( \alpha <: \sigma \), and then we substitute it with an intersection type \( \alpha \& [\sigma]_{\Sigma} \).

\textbf{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. \( [\Sigma] \) elaborates a \( F_< \) context to a \( F_+ \) type context, in which subtyping constraints \( \alpha <: \sigma \) of type variables are elaborated to disjointness constraints \( \alpha + \top \) and all term variables are ignored. \( [\Sigma] \) elaborates a \( F_< \) context to a \( F_+ \) 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 \( \Sigma \), if \( \epsilon \) elaborates to \( E \), then type applications \( \epsilon \sigma \)
elaborates to \( E [\sigma]_\Sigma \). Now let us consider an example.

**Example 17.** Consider a \( F_{\triangleleft} \) judgment

\[
\beta <: \text{Int} \vdash (\Lambda (\alpha <: \top). \lambda (x : \alpha). x) : \beta : \beta
\]

Here the type application type-checks because by rule \( \text{T-sub-top} \) we have \( \beta <: \top \). If we elaborate \( \epsilon \sigma \) to \( E [\sigma]_\Sigma \) directly, the resulting expression is

\[
(\Lambda (\alpha <: \top). (\lambda (x : \alpha) \rightarrow (\alpha <: \top)) (\beta \& \text{Int})
\]

Note that as \( F_{\triangleleft}^+ \) does not have annotated abstractions, we put the elaborated arrow type as the type annotation. Following the typing rule of \( F_{\triangleleft}^+ \), we can infer the type of this expression:

\[
\beta <: \top; \bullet \vdash (\Lambda (\alpha <: \top). ((\lambda (x : \alpha \rightarrow (\alpha <: \top)) (\beta \& \text{Int})) \Rightarrow (\beta \& \text{Int} \& \top) \rightarrow (\beta \& \text{Int} \& \top))
\]

However, the expected result type \( \beta \rightarrow \beta \) elaborates to

\[
(\beta \& \text{Int}) \rightarrow (\beta \& \text{Int})
\]

Now we get a mismatch between the actual type \( ((\beta \& \text{Int} \& \top) \rightarrow (\beta \& \text{Int} \& \top)) \) and the expected type \( ((\beta \& \text{Int}) \rightarrow (\beta \& \text{Int})) \) 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 \( (\beta \& \text{Int}) <: \top \). Therefore we can show that the following two subtyping instances are valid: (1) \( (\beta \& \text{Int} \& \top) \rightarrow (\beta \& \text{Int} \& \top) <:(\beta \& \text{Int} \& \top) \rightarrow (\beta \& \text{Int} \& \top) \); and (2) \( (\beta \& \text{Int}) \rightarrow (\beta \& \text{Int}) <:(\beta \& \text{Int} \& \top) \rightarrow (\beta \& \text{Int} \& \top) \).

More generally, we prove that elaboration commutes with substitution, yielding isomorphic types. Consider that under the context \( \Sigma \), we have a type application \( \epsilon \sigma \), where \( \epsilon \) 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 \( F_{\triangleleft} \) 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., \( [[\sigma/\alpha]](\forall \tau_2)[(\Sigma, \alpha <: \tau_1)] \).

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

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 \( \text{T-sub} \) in kernel \( F_{\triangleleft} \)), 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 \( \text{T-sub} \) in \( F_{\triangleleft}^+ \)), 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:

\[
((\Lambda (\alpha <: \top). (\lambda (x : \alpha) \rightarrow (\alpha <: \top)) (\beta \& \text{Int})) \Rightarrow (\beta \& \text{Int} \rightarrow (\beta \& \text{Int}))
\]

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 \( \text{F-tapp} \), 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.
\( F_{<} \) \( (\Lambda (x : \alpha) \cdot \lambda x : \alpha.1) \rightarrow \lambda x : \text{Int}.1 \)

\( F_{i}^{+} \) \( ((\Lambda (x : \alpha) \cdot (\lambda x : \alpha.1) \rightarrow \text{Int})) \rightarrow \text{Int} \)

\( F_{co} \) \( ((\lambda x : \alpha.1) \rightarrow ((\lambda x : \alpha.1) \rightarrow \text{Int}) \rightarrow \text{Int} \) \( (\lambda x : \text{Int}.1) \rightarrow \text{Int} \)

**Theorem 19** (Type-safety of elaboration). If \( \Sigma \vdash e : \sigma \leadsto E \), then \( \| \Sigma \| ; \| \Sigma \| \vdash E \Rightarrow [\sigma]_{\Sigma} \).

However, due to the implicit upcasting (rule F-SUB), a \( F_{<} \) expression can possibly elaborate to many different ones in \( F_{i}^{+} \). For example, consider \( (\lambda x : \top.2) \). Two elaborations (among others) are (1) \( ((\lambda x : 2) : \top \rightarrow \text{Int}) (1 : \top) \); and (2) \( ((\lambda x : 2) : \top \rightarrow \text{Int}) : \text{Int} \rightarrow \text{Int} ) \). Therefore, we prove that different elaborations lead to contextually equivalent results.\(^3\)

**Theorem 20** (Coherence of elaboration). If \( \triangleright e : \sigma \leadsto E_{1} \), and \( \triangleright e : \sigma \leadsto E_{2} \), and \( \triangleright e : E_{1} \rightarrow E_{2} \Rightarrow [\sigma]_{\circ} \leadsto e_{1} \), and \( \triangleright e : E_{1} \rightarrow E_{2} \Rightarrow [\sigma]_{\circ} \leadsto e_{2} \), then \( \triangleright e : E_{1} \leadsto e_{1} \circ e_{2} \).

We also prove a weaker simulation result\(^4\): if the standard direct operational semantics of kernel \( F_{<} \) produces \( e_{1} \rightarrow e_{2} \), and \( e_{2} \) elaborates to \( E_{2} \) in \( F_{i}^{+} \), which in turn elaborates to \( e_{2} \) in \( F_{co} \), then \( e_{1} \) elaborates to \( E_{1} \) in \( F_{i}^{+} \), which in turn elaborates to \( e_{1} \) in \( F_{co} \), and \( e_{1} \rightarrow e_{1}^{'} \), 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 \( e_{1}^{'} = ((\lambda x : 1) \rightarrow \alpha) (\lambda x : 1) \) that is contextually equivalent to \( e_{2} = \lambda x : 1 \).

**Theorem 21** (Simulation). If \( e_{1} \rightarrow e_{2} \), and \( \triangleright e : \sigma \leadsto E_{2} \), and \( \triangleright e : E_{1} \rightarrow E_{2} \Rightarrow [\sigma]_{\circ} \leadsto e_{2} \), then there exist \( E_{1} \), \( e_{1} \), \( e_{1}^{'} \) such that \( \triangleright e : \sigma \leadsto E_{1} \), and \( \triangleright e : E_{1} \rightarrow E_{2} \Rightarrow [\sigma]_{\circ} \leadsto e_{1} \), and \( e_{1} \rightarrow e_{1}^{'} \), where \( \triangleright e : e_{1}^{'} \circ e_{2} \).

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.

---

\(^3\) 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 \( F_{<} \) 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.

\(^4\) Note that \( \lambda^{1} \) does not provide a semantics [34], so we did not discuss the operational semantics in Section 4. If \( \lambda^{1} \) had a operational semantics, we believe a similar theorem would apply.
6.1 Variants of Row Polymorphism

According to Rémy [53], 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, while the free group does. We have already shown that $F^+_l$ supports $\lambda_!^l$, a calculus in the strict group, with a more fine-grained control as disjointness allows duplicate labels as long as their types are disjoint. $\lambda^{\text{TIR}}$ [60] 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, $\lambda^{\text{TIR}}$ incorporates a kind system that distinguishes rows from types. We believe that $F^+_l$ could also serve as a target for $\lambda^{\text{TIR}}$, as type-indexed rows are closely related to disjoint intersections. Thus an elaboration from $\lambda^{\text{TIR}}$ to $F^+_l$ 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, 53, 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 $F^+_l$ 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]:

\[
inc = \lambda \alpha <: \{l : \text{Int}\}. \lambda (x : \alpha). x \leftarrow \{l = x.l + 1\}
\]

In $F^+_l$, we can define

\[
inc' = \lambda (\alpha \ast \{l : \text{Int}\}). \lambda (x : \alpha \& \{l : \text{Int}\}). (x : \alpha, \{l = (x : \{l : \text{Int}\}).l + 1\})
\]

There are two differences. Firstly, the type arguments to the two functions are different: $inc$ expects a type argument which includes $\{l : \text{Int}\}$, while $inc'$ expects a type argument which excludes $\{l : \text{Int}\}$, and $\{l : \text{Int}\}$ 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 $\text{Int}$, while $inc'$ can accept arguments of type $\{l : \text{Int}\} \& \{l : \text{Bool}\}$. 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 $\lambda_!^l$ or $F^+_l$ 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 [61]. Many other row type systems [53, 64, 63, 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 $F_{<}$ [23] includes a more powerful formulation of subtyping for universal quantification (rule $\text{F-SUB-FORALLALT}$), which is contravariant in the bound types and covariant in the body types. However, this subtyping rule renders subtyping in full $F_{<}$ undecidable [49].

\[
\Sigma \vdash \tau_2 <: \tau_1 \quad \Sigma, \alpha <: \tau_2 \vdash \sigma_1 <: \sigma_2 \\
\Sigma \vdash \forall (\alpha <: \tau_1). \sigma_1 <: \forall (\alpha <: \tau_2). \sigma_2 \quad \text{F-SUB-FORALLALT}
\]
Moreover, this rule breaks the encoding. Consider the example [48]:

\[ \vdash \forall (\alpha <: \top), \alpha <: (\forall (\alpha <: \text{Int}). \alpha) \]

which elaborates to a non-derivable \( F^+_i \) judgment

\[ \vdash (\alpha * \top). \alpha & \top <: (\forall (\alpha * \top). \alpha & \text{Int}) \]

since \( \alpha * \top \vdash \alpha \& \text{Int} \) 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 one type if the first has more instances [45]. For example, the following judgment holds true, as \( \alpha \) can be instantiated to \( \text{Int} \) to get \( \text{Int} \rightarrow \text{Int} \):

\[ \forall \alpha. \alpha \rightarrow \alpha <: \text{Int} \rightarrow \text{Int} \]

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

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 \( F_< \).

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 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 \( F_< \) 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 \( \forall (\alpha <: \text{Int}). \alpha \rightarrow \alpha \), which, based on the encoding, elaborates to \( \forall \alpha. \alpha & \text{Int} \rightarrow \alpha & \text{Int} \). The original type expects a type argument which is a subtype of \( \text{Int} \); while in the intersection type system, the elaborated type can take any type argument, e.g., \( \text{Bool} \), and then expect a term argument of type \( \text{Int} & \text{Bool} \). In intersection type systems (e.g., [43]) where \( \text{Int} & \text{Bool} \) is uninhabited (equivalent to the bottom type), \( f \text{Bool} \) can take nothing. Yet, in calculi with the merge operator, we can have, e.g., \( f \text{Bool}(1, \text{True}) \).

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 \( F_< \) [23] (see also Section 6.2) is proved to be undecidable [49]. The kernel Fun variant [12], which restricts the subtyping of bounds to be invariant, is decidable.

Pierce [48] 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 \( \alpha \) bounded by a type \( \sigma \) can be encoded by replacing every occurrence of \( \alpha \) by \( \beta \land \sigma \) where \( \beta \) is a fresh unbounded variable.
Castagna et al. [17] further mentioned that the possible instantiation of a type variable $\alpha$ with a upper bound $\sigma$ and a lower bound $\tau$ is equivalent to the possible instantiation of $(\tau \lor \beta) \land \sigma$. Dolan and Mycroft [26] used a similar encoding as one of the main ingredients of the unification algorithm: $\alpha <: \sigma^-$ (where types have polarity) implies the bisubstitution $	heta = [(\mu^\beta \alpha \cap [\beta/\alpha^-(\sigma^-)])/\alpha^-]$, which by unrolling implies that $\theta(\alpha^-) = \alpha \cap \theta(\sigma^-)$. 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, 53, 11, 64, 63]. Reynolds [57] 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 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 [57, 55] includes a merge operator. However, several restrictions were imposed to make the merge operator coherent [56]. 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 [48] 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 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, 58]. In those calculi, intersection types only increase the expressiveness of types, but not the expressiveness of terms. For example, the intersection type Int & 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 $(\bot)$.

**Conclusion and Future Work**

We have presented the elaboration from kernel $F_<$ and $\lambda^\updownarrow$ 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 $F_<$ identifies one encodable fragment of $F_<$, and thus validates the previous informal observation by Pierce. The elaboration from $\lambda^\updownarrow$ 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 $F_<$, as well as other row calculi. We also plan to study type inference of $F^+_i$. 
References


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


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


