Disjoint Polymorphism

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi

The University of Hong Kong
{zyshi,bruno}@cs.hku.hk

Abstract. The combination of *intersection types*, a *merge operator* and *parametric polymorphism* enables important applications for programming. However such combination makes it hard to achieve the desirable property of a *coherent semantics*: all valid reductions for the same expression should have the same value. Recent work proposed *disjoint intersections types* as a means to ensure coherence in a simply typed setting. However, the addition of parametric polymorphism was not studied.
This paper presents F_{i}: a calculus with *disjoint intersection types*, a variant of *parametric polymorphism* and a *merge operator*. F_{i} is both type-safe and coherent. The key difficulty in adding polymorphism is that, when a type variable occurs in an intersection type, it is not statically known whether the instantiated type will be disjoint to other components of the intersection. To address this problem we propose *disjoint polymorphism*: a constrained form of parametric polymorphism, which allows programmers to specify disjointness constraints for type variables. With disjoint polymorphism the calculus remains very flexible in terms of programs that can be written, while retaining coherence.

1 Introduction

Intersection types [11,23] are an increasingly popular language feature for modern programming languages, such as Microsoft's TypeScript [4], Redhat's Ceylon [1], Facebook's Flow [3] and JOAO: cited websites Scala [19]. In those languages a typical use of intersection types, which has been known for a long time [] JOAO: I could not find the first use, is to model the subtyping aspects of OO-style multiple inheritance. For example, the following Scala declaration:

```
class A extends B with C
```

says that the class `A` implements *both* `B` and `C`. The fact that `A` implements two interfaces/traits is captured by an intersection type between `B` and `C` (denoted in Scala by `B with C`). Unlike a language like Java, where `implements` (which plays a similar role to `with`) would be a mere keyword, in Scala intersection types are first class. For example, it is possible to define functions such as:

```
def narrow(x : B with C) : B = x
```

taking an argument with an intersection type `B with C`.

The existence of first-class intersections has lead to the discovery of other interesting applications of intersection types. For example, TypeScript's documentation motivates intersection types[1] as follows:

> *You will mostly see intersection types used for mixins and other concepts that don't fit in the classic object-oriented mold. (There are a lot of these in JavaScript!)*

Two points are worth emphasizing. Firstly, intersection types are being used to model concepts that are not like the classical (class-based) object-oriented programming. Indeed, being a prototype-based language, JavaScript has a much more dynamic notion of object composition compared to class-based languages: objects are composed at run-time, and their types are not necessarilly statically known. Secondly, the use of intersection types in TypeScript is inspired by common programming patterns in the (dynamically typed) JavaScript. This hints that intersection types are useful to capture certain programming patterns that are out-of-reach for more conventional type systems without intersection types.

Central to TypeScript's use of intersection types for modelling such a dynamic form of mixins is the function:

```
function extend<T, U>(first: T, second: U) : T & U {...}
```

The name *extend* is given as an analogy to the *extends* keyword commonly used in OO languages like Java. The function takes two objects (`first` and `second`) and produces an object with the intersection of the types of the original objects. The implementation of *extend* relies on low-level (and type-unsafe) features of JavaScript. When a method is invoked on the new object resulting from the application of `extend`, the new object tries to use the `first` object to answer the method call and, if the method invocation fails, it then uses the `second` object to answer the method call.

The *extend* function is essentially an encoding of the *merge operator*. The merge operator is used on some calculi [26,25,9,13,27] as an introduction form for intersection types. Similar encodings to those in TypeScript have been proposed for Scala to enable applications where the merge operator also plays a fundamental role [20]. JOAO: any other uses? Unfortunately, the merge operator is not directly supported by TypeScript, Scala, Ceylon or Flow. There are two possible reasons for such lack of support. One reason is simply that merge operator is not well-known: many calculi with intersection types in the literature do not have explicit introduction forms for intersection types. The other reason is that, while powerful, the merge operator is well-known to introduce *(in)coherence* problems [25,13]. If care is not taken, certain programs using the merge operator do not have a unique semantics, which significantly complicates reasoning about programs.

Solutions to the problem of coherence in the presence of a merge operator exist for simply typed calculi [26,25,9,27], but no prior work addresses polymorphism. Most recently Oliveira et al. [27] proposed using *disjoint intersection*

[1] `https://www.typescriptlang.org/docs/handbook/advanced-types.html`

types to guarantee coherence in a calculus with intersection types and a merge operator. The key idea is to allow only disjoint types in intersections. If two types are disjoint then there is no ambiguity in selecting a value of the appropriate type from an intersection, guaranteeing coherence.

Combining parametric polymorphism with disjoint intersection types, while retaining enough flexibility for practical applications, is non-trivial. The key issue is that when a type variable occurs in an intersection type it is not statically known whether the instantiated type will be disjoint to other components of the intersection. A naive way to add polymorphism is to forbid type variables in intersections, since they may be instantiated with a type which is not disjoint to other types in an intersction. Unfortunatelly this is too conservative and prevents many useful programs, including the `extend` function, which uses an intersection of two type variables `T` and `U`.

This paper presents $\mathsf{F}_\mathtt{i}$: a core calculus with *disjoint intersection types*, a variant of *parametric polymorphism* and a *merge operator*. The key innovation in the calculus is *disjoint polymorphism*: a constrained form of parametric polymorphism, which allows programmers to specify disjointness constraints for type variables. With disjoint polymorphism the calculus remains very flexible in terms of programs that can be written with intersection types, while retaining coherence. In $\mathsf{F}_\mathtt{i}$ the `extend` function is implemented as follows:

```
let extend T (U * T) (first : T, second : U) : T & U = first ,, second
```

From the typing point of view, the difference between `extend` in TypeScript and $\mathsf{F}_\mathtt{i}$ is that the type variable `U` now has a *disjointness constraint*. The notation `U * T` means that the type variable U can be instantiated to any types which are disjoint to the type `T`. Unlike TypeScript, the definition of `extend` is trivial, type-safe and guarantees coherence by using the built-in merge operator (`,,`).

The applicability of $\mathsf{F}_\mathtt{i}$ is illustrated with examples using `extend` ported from TypeScript, and various operations on polymorphic extensible records [18,14,16]. The operations on polymorphic extensible records show that $\mathsf{F}_\mathtt{i}$ can encode various operations/application of row types [28]. However, in contrast to various existing proposals for row types and extensible records, $\mathsf{F}_\mathtt{i}$ supports general intersections and not just record operations.

$\mathsf{F}_\mathtt{i}$ and the proofs of coherence and type-safety are formalized in the Coq theorem prover [2]. The proofs are complete except for a minor (and trivially true) variable renaming lemma used to prove the soundness between two subtyping relations used in the formalization. The problem arizes from the combination of the locally nameless representation of binding [5] and existential quantification, which prevents a Coq proof for that lemma.

In summary, the contributions of this paper are:

- **Disjoint Polymorphism:** A novel form of universal quantification where type variables can have disjointness constraints. Disjoint Polymorphism enables a flexible combination of intersection types, the merge operator and parametric polymorphism.

- **Coq Formalization of F_i and Proof of Coherence:** An elaboration semantics of System F_i into System F is given. Type-soundness and coherence are proved in Coq [2].
- **Applications:** We show how F_i provides basic for dynamic mixins and various operations on polymorphic extensible records.

2 Overview

This section introduces F_i and its support for intersection types, parametric polymorphism and the merge operator. It then discusses the issue of coherence and shows how the notion of disjoint intersection types and disjoint quantification achieves a coherent semantics. This section uses some syntactic sugar, as well as standard programming language features, to illustrate the various concepts in F_i. Although the minimal core language that we formalize in Section 4 does not present all such features, such syntactic sugar is trivial to add.

2.1 Intersection Types and the Merge Operator

Intersection types. The intersection of type A and B (denoted as `A & B` in F_i) contains exactly those values which can be used as both values of type A and of type B. For instance, consider the following program in F_i:

```
let x : Int & Bool = ... in -- definition omitted
let succ (y : Int) : Int = y+1 in
let not (y : Bool) : Bool = if y then False else True in (succ x, not x)
```

If a value `x` has type `Int & Bool` then `x` can be used anywhere where either a value of type `Int` or a value of type `Bool` is expected. This means that, in the program above the functions `succ` and `not` – simple functions on integers and booleans, respectively – both accept `x` as an argument.

Merge operator. The previous program deliberately ommitted the introduction of values of an intersection type. There are many variants of intersection types in the literature. Our work follows a particular formulation, where intersection types are introduced by a *merge operator* [26,25,9,13,27]. As Dunfield [13] has argued a merge operator adds considerable expressiveness to a calculus. The merge operator allows two values to be merged in a single intersection type. For example, an implementation of `x` in F_i is `1,,True`. Following Dunfield's notation the merge of v_1 and v_2 is denoted as $v_1,,v_2$.

2.2 Coherence and Disjointness

Coherence is a desirable property for a semantics. A semantics is coherent if any *valid program* has exactly one meaning [25] (that is, the semantics is not ambiguous). Unfortunately the implicit nature of elimination for intersection types built

[2] **Note to reviewers:** Due to the anonymous submission process, the Coq formalization is submitted as supplementary material.

with a merge operator can lead to incoherence. This is due to intersections with overlapping types, as in `Int&Int`. The result of the program (`(1,,2) : Int`) can be either `1` or `2`, depending on the implementation of the language.

Disjoint intersection types One option to restore coherence is to reject programs which may have multiple meanings. The λ_i calculus [27] – a simply-typed calculus with intersection types and a merge operator – solves this problem by using the concept of disjoint intersections. The incoherence problem with the expression $1,,2$ happens because there are two overlapping integers in the merge. Generally speaking, if both terms can be assigned some type C then both of them can be chosen as the meaning of the merge, which in its turn leads to multiple meanings of a term. Thus a natural option is to forbid such overlapping values of the same type in a merge. In λ_i intersections such as `Int&Int` are forbidden, since the types in the intersection overlap (i.e. they are not disjoint). However an intersection such as `Char&Int` is ok because the set of characters and integers are disjoint to each other.

2.3 Parametric Polymorphism

Unfortunately, combining parametric polymorphism with disjoint intersection types is non-trivial. Consider the following program (uppercase Latin letters to denote type variables):

```
let merge3 A (x : A) : A & Int = x,,3 in
```

The `merge3` function takes an argument `x` of some type (`A`) and merges `x` with `3`. Thus the return type of the program is `A & Int`. `merge3` is unproblematic for many possible instantiations of `A`. However, if `merge3` instantiates `A` with a type that overlaps (i.e. is not disjoint) with `Int`, then incoherence may happen. For example:

```
merge3 Int 2
```

can evaluate to both 2 or 3.

Forbidding type variables in intersections A naive way to ensure that only programs with disjoint types are accepted is simply to forbid type variables in intersections. That is, an intersection type such as `Char&Int` would be accepted, but an intersection such as `A&Int` (where `A` is some type variable) would be rejected. The reasoning behind this design is that type variables can be instantiated to any types, including those already in the intersection. Thus forbidding type variables in the intersection will prevent invalid intersections arizing from instantiations with overlapping types.

Such design does guarantee coherence and would prevent `merge3` from type-checking. Unfortunatelly the big drawback is that the design is too conservative and many other (useful) programs would be rejected. In particular, the `extend` function from Section 1 would also be rejected.

Other approaches Another option to mitigate the issues of incoherence, without using disjoint intersection types is to allow for a biased choice: that is multiple values of the same type may exist in an intersection, but an implementation gives preference to one of them. The encodings of merge operators in TypeScript and Scala [20] use such an approach. A first problem with this approach, which has already been pointed out by Dunfield [13], is that the choice of the corresponding value is tied up to a particular choice in the implementation. In other words incoherence still exists at the semantic level, but the implementation makes it predictable which overlapping value will be chosen. From the theoretical point-of-view it would be much better to have a clear, coherent semantics, which is independent from concrete implementations. Another problem is that the interaction between biased choice and polymorphism can lead to counter-intuitive programs, since instantiation of type-variables affects the type-directed lookup of a value in an intersection.

2.4 Disjoint Polymorphism

To avoid being overly conservative, while still retaining coherence in the presence of parametric polymorphism and intersection types, F_i uses *disjoint polymorphism*. Inspired by bounded quantification [7], where a type variable is constrained by a type bound, disjoint polymorphism allows type variables to be constrained so that they are disjoint to some given types.

With disjoint quantification a variant of the program `merge3`, which is accepted by F_i, is written as:

```
let merge3 ( A * Int ) (x : A) : A & Int = x,,3 in
```

In this variant the type `A` can be instantiated to any types disjoint to `Int`. Such restriction is expressed by the notation `A * Int`, where the left-side of `*` denotes the type variable being declared (`A`), and the right-side denotes the disjointness constraint (`Int`). For example,

```
merge3 Bool True
```

is accepted. However, instantiating `A` with `Int` fails to type-check.

Multiple constraints Disjoint quantifications allows for multiple constraints. For example, in the following variant of the `merge3` there is an additional boolean in the merge:

```
let merge3b ( A * Int & Bool ) (x : A) : A & Int & Bool = x,,3,,True in
```

In this case the type variable `A` needs to be disjoint to both `Int` and `Bool`. In F_i such constraint is specified using an intersection type `Int & Bool`. In general, multiple constraints are specified by creating an intersection of all required constraints.

Type variable constraints Disjoint quantification also allows type variables to be disjoint to previously defined type variables. For example, the following program is accepted by F_i:

```
let fst A ( B * A ) (x: A & B) : A = x
in ...
```

The program has two type variables `A` and `B`. `A` is unconstrained and can be instantiated with any type. However, the type variable `B` can only be instantiated with types that are disjoint to `A`. The constraint on `B` ensures that the intersection type `A & B` is disjoint for all valid instantiations of A and B. In other words, only coherent uses of `fst` will be accepted. For example, the following use of `fst`:

```
fst Int Char (1,,'c')
```

is accepted since `Int` and `Char` are disjoint, thus satisfying the constraint on the second type parameter of `fst`. Furthermore, problematic uses of `fst`, such as:

```
fst Int Int (1,,2)
```

are rejected because `Int` is not disjoint with `Int`, thus failing to satisfy the disjointness constraint on the second type parameter of `fst`.

Empty constraint The type variable A in the `fst` function has no constraint. In F_i this actually means that A should be associated with the empty contraint, which raises the question: which type should be used to represent such empty constraint? Or, in other words, which type is disjoint to every other type? It is obvious that this type should be one of the bounds of the subtyping lattice: either $\bot$ or $\top$. The essential intuition here is that the more specific a type in the subtyping relation is, the less types exist that are disjoint to it. For example, `Int` is disjoint to all types except the intersections that contain `Int`, `Int` itself, and $\bot$; while `Int`&`Char` is disjoint to all types that `Int` is, plus the types disjoint to `Char`. Thus, the more specific a type variable constraint is, the less options we have to instantiate it with. This reasoning implies that $\top$ should be treated as the empty constraint. Indeed, in F_i, a single type variable A is only syntactic sugar for $\mathsf{A} * \top$.

2.5 Stability of Substitutions

From the technical point of view, the main challenge in the design of F_i is that, *in general, types are not stable under substitution.* This contrasts, for example, with System F where types are stable under substitution. That is in System F the following property (among others) holds:

Lemma 1 (Stability of Substitution). *For any well-formed types* A *and* B*, and a type variable* α*, the result of substituting* α *for* A *in* B *is also a well-formed type.*

In F_i if a type variable A is substituted in a type T_1, for a type T_2 (written $[\mathsf{A} := \mathsf{T}_2]\ \mathsf{T}_1$), where T_1 and T_2 are well-formed, the resulting type might be ill-formed. To understand why, recall the previous example:

```
fst Int Int (1,,2)
```

The type signature of `fst` may be read as $\forall A(B * A).(A\&B) \to A$. An application to the type `Int` will lead to instantiation of the variable A, leading to the type $\forall(B * \texttt{Int}).(\texttt{Int}\&B) \to \texttt{Int}$. Now, the second `Int` application is problematic, since instantiating B with `Int` will lead to the ill-formed type $(\texttt{Int}\&\texttt{Int}) \to \texttt{Int}$. However, from this example it is easy to see that all types which are not problematic are exactly the ones disjoint with A. This paper shows how a weaker version of the usual type substituition stability still holds, namely by requiring that the type varible's disjointness constraint is compatible with the type as target of the instantiation.

3 Applications

To illustrate the applicability of $\mathsf{F}_\mathtt{i}$ we show two applications. The first application shows how to mimic TypeScript's documentation example of dynamic mixins in $\mathsf{F}_\mathtt{i}$. The second application shows how $\mathsf{F}_\mathtt{i}$ enables a powerful form of polymorphic extensible records.

3.1 Dynamic Mixins

TypeScript is a language that adds static type checking to JavaScript. Its type-system is structural, even though at a first glance it might resemble nominal languages such as Java or C#. Amongst numerous static typing constructs, TypeScript supports a form of intersection types, with no merge operator. However, it is possible to define a function that mimics the merge operator, as some form of mixin composition:

```
function extend<T, U>(first: T, second: U): T & U {
  let result = <T & U>{};
  for (let id in first) {
    (<any>result)[id] = (<any>first)[id];
  }
  for (let id in second) {
    if (!result.hasOwnProperty(id)) {
      (<any>result)[id] = (<any>second)[id];
    }
  }
  return result;
}

class Person { constructor(public name : string, public male : boolean)
    { } }
interface Loggable { log() : void; }
class ConsoleLogger implements Loggable {
    log() {
      // ...
    }
}
var jim = extend(new Person("Jim"), new ConsoleLogger());
```

```
var n = jim.name;
jim.log();
```

In this example, taken from TypeScript's documentation, an `extend` function is defined for mixin composition, and two classes `Person` and `ConsoleLogger`. Two instaces of those classes are then composed in a variable `jim` with the type of the intersection of both, where it is type-safe to access both the `name` property from `Persion` and `ConsoleLogger`.

This definition relies essentially on biased choice. Let us take a closer look at the definition of `extend`. Given two arguments `first` of type `T`, and `second` of type `U`, it returns an intersection of both types represented as `T \inter U`. The function starts by creating a variable `result` with the type of the intersection, It then iterates through the `first`'s properties and copies them to the `result`. Next, it iterates through the `second`'s properties but it only copies the properties that `result` does not possess (i.e. the ones present in `first`). This means that the implementation is left-biased, as the properties of left type of the intersection are chosen in favor of the ones present in the right. However, in TypeScript this may be a cause of severe problems since that, at the time of writing, intersections at type-level are actually right-biased! For example, the following code is well-typed:

```
class Dog { constructor(public name : string, public male : string) { } }
var fool : Dog & Person = extend(new Dog("Pluto","non-sense"),new
    Person("Arnold",true))
boolean b = fool.male
```

Note how `fool.male`, at run-time, will contain a value of type `String`!

Other problematic issues regarding the semantics of intersection types can include the order of the types in an intersection, or even intersections including repeated types. This motivates the need to define a clear meaning to the practical application of intersection types.

In F_i, the merge combinator is directly embedded in the semantics of the language, and thus there is no need to define such combinators. In fact, the introduction of disjoint intersection types can be seen as a type-safe solution for all of the mentioned problems involving intersection types. JOAO: mention the diamond problem? and that two fields with same name can be composed as long as their types are disjoint For the previous TypeScript examples, assuming a straightforward translation from objects to (polymorphic) records, then the intersection `Person & ConsoleLogger` would be well-typed in F_i. However, the intersection `Person & Dog` would be ill-typed, since the types share the same field `name`. We will next explore the enconding of polymorphic records in greater depth.

3.2 Extensible Records

Our system can be used to polymorphic extensible records. Describing and implementing records within programming languages is certainly not novel and has been extensively studied in the past, including systems with row types [28,29];

predicates [15,14]; flags [24]; conditional constraints; and others. BRUNO: references here; and give names to a few approaches (row types for example). Most of the systems are entirely focused on concrete aspects of records (i.e. expressiveness, compilation, etc), while ours specializes the more general notion of intersection types. In this section we aim at comparing our approach with such systems.

Systems with records usually rely on 3 basic operations: selection, restriction and extension/concatenation. We will first introduce the basic syntax and typing of records, then the basic operations of records in the context of F_i and finally we will discuss their expressivity in comparison to other record systems.

Record terms and types Our system directly encodes a term for the single record construct $\{l = e\}$, where l is some label and e is some other term. This term comes with its associated type, denoted as $\{l : T\}$, where T is a type that is attributed to e.

Selection The select operator is directly embedded in our language. It follows the usual syntax of $e.l$, where e is an expression of type $\{l : \alpha\}$ and l is a label. A polymorphic function which extracts any record that include the label l of type α could be written as:

```
let select A (r : {l : A}) : A = r.l in ...
```

Note how, through the use of subtyping, this function will accept any intersection type that contains the single record $\{l : A\}$. This may resemble other systems with subtyping [8,22], although it is slightly more general – the type A is not restricted only restricted to record types.

Restriction In constrast with most systems, restriction is not directly embedded on our language. Instead, we can make use of subtyping to define such operator:

```
let remove A (B * {l : A}) (x : { l : A } & r) : r = x in ...
```

Extension/Concatenation The most usual operators for combining records are extension and concatenation. Even though that in some systems, the latter is defined in terms of the former, languages that opt to include concatenation usually rely on specific semantics for it. JOAO: add references Our system is suitable for encoding both of these operations, but we argue that concatenation is the natural primitive operator, due to the resemblance with our merge operator. Indeed, [14] also define a *merge* operator, which is quite similar to our *merge* for intersection types, except it envoles only record types. For instance, in their system, the following function concatenates a single record with field `l` of type `Int` with another record that lacks this field [3]:

```
let addL_l (A # l) (x : A) : (a || { l : Int }) = ... in ...
```

The reader might notice the resemblance with our system:

[3] The syntax is slightly modified with respect to the original notation

```
let addL2 (A * { l : Int }) (x : A) : (A & { l : Int }) = ... in ...
```

This shows that one can use disjoint quantification to express negative field information, which is very close to the system described in [14]. Note how one has to explicitly state the type of the constraint in addL_2, whereas addL_1 does not require this. The same generality of disjoint intersection types that allows one to encode record types is the one that forces us to add this extra type in the constraint. However, there is a slight gain with this approach: addL_2 accepts more types than addL_1. Namely, all (intersection) types that contain label `l`, with a field type *disjoint* to `Int`. Had one meant to forbid records with *any* `l` fields, then one could write:

```
let addL3 (A * { l : ⊥ }) (x : A) : A & { l : ⊥ } = ... in ...
```

Unfortunately, our system does not support the $\bot$ type, so this program would not be accepted. We will get into the reasons for this in further detail, in Section 5.

Other systems with record concatenation usually define predicates, in terms of field absence or presence (with a type α). This raises the question: how would one classify our system in terms of extension? As noted in [**?**], systems typically can be categorized into two distinct groups in what concerns extension: the strict and the free. The former does not allow field overriding when extending a record (i.e. one can only extend a record with a field that is not present in it); while the latter does account for field overriding. Our system can be seen as hybrid of these two kinds of systems. Next we will show a comparison in terms of expressability between F_i and other systems with records that hopefully will enlighten the reader on this matter.

Expressibility JOAO: maybe the average function is too long? we can change to a type with only one field In [16] – a strict system with extension – an example of a function that uses record types is the following:

```
let average1 (R\y, R\y) => (r : { R | x : Int, y : Int }) : Int = (r.x +
    r.y) / 2
in ...
```

The type signature says that for any record with type `r`, that lacks both `x` `y`, can be accepted as parameter extended with `x` `y`, returning an integer. Note how the bounded polymorphism is essential to ensure that `r` does not contain `x` nor `y`. On the other hand, in a system with free extension as in [18], the more general program would be accepted:

```
let average2 R x y (r : { x : Int, y : Int | R }) : Int = (r.x + r.y) / 2
in ...
```

In this case, if `r` contains either field `x` or field `y`, they would be shadowed by the labels present in the type signature. In other words, if a record with multiple `x` fields, the most recent (i.e. left-most) would be used in any function which accesses `x`. JOAO: add example of a system using subtyping?

In F_i, this program could be re-written as [4]:

```
let average3 (R * { x : Int } & { y : Int }) (r : { x : Int } & { y :
    Int } & R) : Int = (r.x + r.y) / 2
in ...
```

Thus more types are accepted this function than in the first system, but less than the second. Another major difference between F_i and the two other mentioned systems, is the ability to combine records with arbitrary types. Our system does not account for well-formedness of record types as the other two systems do (i.e. using a special *row* kind), since our encoding of records piggybacks on the more general notion of disjoint intersection types.

Finally, it is also worth noting that systems using subtyping may suffer from the so-called *update* problem. JOAO: show example (for both update problems?) F_i does not suffer from this problem. JOAO: since we have no refinement types? We may illustrate this by defining a suitable update function, in a similar fashion to [18]:

```
let update A B (R * { l : A }) (r : { l : A } & R) (v : B) : { l : B } &
    R = { l = v } ,, (remove a r x)
in ...
```

4 The F_i Calculus

This section presents the syntax, subtyping, and typing of F_i: a calculus with intersection types, parametric polymorphism, records and a merge operator. This calculus is an extension of λ_i and Dunfield's calculus [13], which are simply typed calculus with intersection types and a merge operator. The novelty of F_i is the addition of *disjoint polymorphism*: a form of parametric polymorphism with disjointness contraints, which allows flexibility while at the same time retaining coherence. Section 6 introduces the necessary changes to the definition of disjointness presented by Oliveira et al. [] in order to add disjoint polymorphism.

All the meta-theory of F_i has been mechanized in Coq, which is available in the supplementary materials submitted with the paper.

4.1 Syntax

The syntax of F_i (with the differences to λ_i highlighted in gray) is:

Types	A, B	$::= \top \mid \texttt{Int} \mid A \to B \mid A \& B \mid \alpha \mid \forall(\alpha * A).\, B \mid \{l : A\}$
Terms	e	$::= \top \mid i \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid e_1 ,, e_2 \mid \Lambda(\alpha * A).\, e \mid e\ A \mid \{l = e\}$
		$\mid e.l$
Contexts	Γ	$::= \cdot \mid \Gamma, \alpha * A \mid \Gamma, x : A$

[4] We do not support exactly this function definition style; however the type signature and expression (modulo infix operators) are exactly as one would write them in F_i

Types. Metavariables A, B range over types. Types include all constructs in λ_i: a top type $\top$; the type of integers $\texttt{Int}$; function types $A \to B$; and intersection types $A\&B$. The main novelty are two standard constructs of System F used to support polymorphism: type variables α and disjoint (universal) quantification $\forall(\alpha{*}A).\,B$. Unlike traditional universal quantification, the disjoint quantification includes a disjointness constraint associatted to a type variable α. Finally, F_i also includes singleton record types, which consist of a label l and an associated type A. We will use $[\alpha := A]\ B$ to denote the capture-avoiding substitution of A for α inside B and $\mathsf{ftv}(\cdot)$ for sets of free type variables.

Terms. Metavariables e range over terms. Terms include all constructs in λ_i: a canonical top value $\top$; integer literals i; variables x, lambda abstractions $(\lambda x.\,e)$; applications $(e_1\ e_2)$; and the *merge* of terms e_1 and e_2 denoted as $e1,,e2$. Terms are extended with two standard constructs in System F: abstraction of type variables over terms $\Lambda(\alpha * A).\,e$; and and application of terms to types $e\ A$. The former also includes an extra disjointness constraint tied to the type variable α, due to disjoint quantification. Singleton records consists of a label l and an associated term e. Finally, the accessor for a label l in term e is denoted as $e.l$.

Contexts. Typing contexts Γ track bound type variables α with disjointness constraints A; and variables x with their type A. We will use $[\alpha := A]\ \Gamma$ to denote the capture-avoiding substitution of A for α in the co-domain of Γ where the domain is a type variable (i.e. substitution in all disjointness constraints). Throughout this paper, we will assume that all contexts are well-formed. Importantly, besides usual well-formedness conditions, in well-formed contexts type variables must not appear free within its own disjointness constraint.

Syntactic sugar BRUNO: Moned some text here: talk about syntactic sugar here. JOAO: should we say anything here about this going against our previous top-disjointness formulation? BRUNO: yes, but not here. We can discuss this in later sections when discussing the technical details. For instance, the type of the identity function in System F that reads $\forall A.A \to A$ is equivalent to the F_i's type $\forall(A * \top).A \to A$.

4.2 Subtyping

The subtyping rules of the form $A <: B$ are shown in Figure 1. At the moment, the reader is advised to ignore the gray-shaded part in the rules, which will be explained later. The following rules are ported from λ_i: S$\top$, S$\mathbb{Z}$, S$\to$, S&R, S&L$_1$ and S&L$_2$.

Polymorphism and Records. The subtyping rules introduced by F_i refer to polymorphic constructs and records. Sα defines subtyping as a reflexive relation on type variables. In S$\forall$ a universal quantifier ($\forall$) is covariant in its body, and contravariant in its disjointness constraints. The SREC rule says that records

$$\boxed{A \text{ ordinary}}$$

$$\texttt{Int} \text{ ordinary} \qquad A \to B \text{ ordinary} \qquad \alpha \text{ ordinary} \qquad \forall(\alpha * B).\, A \text{ ordinary}$$

$$\{l : A\} \text{ ordinary}$$

$$\boxed{A <: B \hookrightarrow E}$$

$$\frac{}{A <: \top \hookrightarrow \lambda x.\, ()}\ S\top \qquad \frac{A_1 <: A_2 \hookrightarrow E_1 \qquad A_1 <: A_3 \hookrightarrow E_2}{A_1 <: A_2 \& A_3 \hookrightarrow \lambda x.\, (E_1\ x, E_2\ x)}\ S\&R$$

$$\frac{}{\texttt{Int} <: \texttt{Int} \hookrightarrow \lambda x.\, x}\ S\mathbb{Z} \qquad \frac{A_1 <: A_3 \hookrightarrow E \qquad A_3 \text{ ordinary}}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda x.\, [\![A_3]\!]_{(E\ (\texttt{proj}_1\ x))}}\ S\&L_1$$

$$\frac{}{\alpha <: \alpha \hookrightarrow \lambda x.\, x}\ S\alpha \qquad \frac{A_2 <: A_3 \hookrightarrow E \qquad A_3 \text{ ordinary}}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda x.\, [\![A_3]\!]_{(E\ (\texttt{proj}_2\ x))}}\ S\&L_2$$

$$\frac{A <: B \hookrightarrow E}{\{l : A\} <: \{l : B\} \hookrightarrow E}\ SRec \qquad \frac{B_1 <: A_1 \hookrightarrow E_1 \qquad A_2 <: B_2 \hookrightarrow E_2}{A_1 \to A_2 <: B_1 \to B_2 \hookrightarrow \lambda f.\, \lambda x.\, E_2\ (f\ (E_1\ x))}\ S\to$$

$$\frac{B_1 <: B_2 \hookrightarrow E_1 \qquad A_2 <: A_1 \hookrightarrow E_2}{\forall(\alpha * A_1).\, B_1 <: \forall(\alpha * A_2).\, B_2 \hookrightarrow \lambda f.\, \Lambda\alpha.\, E_1\ (f\ \alpha)}\ S\forall$$

Fig. 1. Subtyping rules of F_i.

are covariant within their fields' types. Also, since the `ordinary` conditions on two of the intersection rules are necessary to produce unique coercions [**?**], the `ordinary` relation needed to be extended. As shown at the top of Figure 1, the new types that compose this unary relation are type variables, universal quantifiers and record types.

Properties of Subtyping. The subtyping relation is reflexive and transitive.

Lemma 2 (Subtyping reflexivity). *For any type* A, $A <: A$.

Proof. By induction on A.

Lemma 3 (Subtyping transitivity). *If* $A <: B$ *and* $B <: C$, *then* $A <: C$.

Proof. By double induction on both derivations.

4.3 Typing

Well-formedness The well-formedness rules are shown in the top part of Figure 2. The new rules over λ_i are WFα and WF$\forall$. Their definition is quite straightforward, but note how we ensure the well-formedness of the constraint in the latter.

BRUNO: I think the figures are quite generous in terms of space at the moment. We can reduce space in between rules. Talk to Linus, he has experience doing this.

Typing rules Our typing rules are formulated as a bi-directional type-system. Just as in λ_i, this ensures the type-system is not only syntax-directed, but also that there is no type ambiguity: that is inferred types are unique. The typing rules are shown in the bottom part of Figure 2. Again, the reader is advised to ignore the gray-shaded part here, as these parts will be explained later. The typing judgements are of the form: $\Gamma \vdash e \Leftarrow A$ and $\Gamma \vdash e \Rightarrow A$. They read: "in the typing context Γ, the term e can be checked or inferred to type A", respectively. The rules that are ported from λ_i are the check rules for $\top$ (T-Top), integers (T-Int), variables (T-Var), application (T-App), merge operator (T-Merge), annotations (T-Ann); and infer rules for lambda abstractions (T-Lam), and the subsumption rule (T-Sub).

Disjoint quantification The new rules, inspired by System F, are the infer rules for type application T-TApp, and for type abstraction T-BLam. Type abstraction is introduced by the big lambda $\Lambda(\alpha * A).\, e$, eliminated by the usual type application $e\ A$ (T-TApp). The disjointness constraint is added to the context in T-BLam. During a type application, the type system makes sure that the type argument satisfies the disjointness constraint. Type application performs an extra check ensuring that the type to be instantiated is compatible (i.e. disjoint) with the constraint associated with the abstracted variable. This is important, as it will retain the desired coherence of our type-system. For ease of discussion, also in T-BLam, we require the type variable introduced by the quantifier to be fresh. For programs with type variable shadowing, this requirement can be met straighforwardly by variable renaming.

Records Finally, T-Rec and T-ProjR deal with record types. The former infers a type for a record with label l if it can infer a type for the inner expression; the latter says if one can infer a record type $\{l : A\}$ from an expression e, then it is safe to access the field l, and infering type A.

5 Disjointness

Section 4 presented a type system with disjoint intersection types and disjoint quantification. In order to prove both type-safe and coherence (in Section **??**), it is necessary to first introduce a notion of disjointness, considering polymorphism and disjointness quantification. This section presents an algorithmic set

$$\boxed{\Gamma \vdash A}$$

$$\frac{}{\Gamma \vdash \texttt{Int}}\ \text{WF}\mathbb{Z} \qquad \frac{\alpha * A \in \Gamma}{\Gamma \vdash \alpha}\ \text{WF}\alpha \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \text{WF}\to \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \{l : A\}}\ \text{WFRec}$$

$$\frac{}{\Gamma \vdash \top}\ \text{WF}\top \qquad \frac{\Gamma \vdash A \qquad \Gamma, \alpha * A \vdash B}{\Gamma \vdash \forall(\alpha * A).\, B}\ \text{WF}\forall \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad \Gamma \vdash A * B}{\Gamma \vdash A\&B}\ \text{WF}\&$$

$$\boxed{\Gamma \vdash e \Rightarrow A \hookrightarrow E \qquad e \text{ synthesizes type } A}$$

$$\frac{}{\Gamma \vdash \top \Rightarrow \top \hookrightarrow ()}\ \text{T-Top} \qquad \frac{}{\Gamma \vdash i \Rightarrow \texttt{Int} \hookrightarrow i}\ \text{T-Int}$$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x \Rightarrow A \hookrightarrow x}\ \text{T-Var} \qquad \frac{\Gamma \vdash e \Leftarrow A \hookrightarrow E}{\Gamma \vdash e : A \Rightarrow A \hookrightarrow E}\ \text{T-Ann}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A_1 \to A_2 \hookrightarrow E_1 \qquad \Gamma \vdash e_2 \Leftarrow A_1 \hookrightarrow E_2}{\Gamma \vdash e_1\ e_2 \Rightarrow A_2 \hookrightarrow E_1\ E_2}\ \text{T-App}$$

$$\frac{\Gamma \vdash e \Rightarrow \forall(\alpha * B).\, C \hookrightarrow E \qquad \Gamma \vdash A \qquad \boxed{\Gamma \vdash A * B}}{\Gamma \vdash e\ A \Rightarrow [\alpha := A]\ C \hookrightarrow E\ |A|}\ \text{T-TApp}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow A \hookrightarrow E_1 \qquad \Gamma \vdash e_2 \Rightarrow B \hookrightarrow E_2 \qquad \Gamma \vdash A * B}{\Gamma \vdash e_1, , e_2 \Rightarrow A\&B \hookrightarrow (E_1, E_2)}\ \text{T-Merge}$$

$$\frac{\Gamma \vdash e \Rightarrow A \hookrightarrow E}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\} \hookrightarrow E}\ \text{T-Rec} \qquad \frac{\Gamma \vdash e \Rightarrow \{l : A\} \hookrightarrow E}{\Gamma \vdash e.l \Rightarrow A \hookrightarrow E}\ \text{T-ProjR}$$

$$\frac{\Gamma \vdash A \qquad \Gamma, \alpha * A \vdash e \Rightarrow B \hookrightarrow E \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda(\alpha * A).\, e \Rightarrow \forall(\alpha * A).\, B \hookrightarrow \Lambda\alpha.\, E}\ \text{T-BLam}$$

$$\boxed{\Gamma \vdash e \Leftarrow A \hookrightarrow E \qquad e \text{ checks against given type } A}$$

$$\frac{\Gamma \vdash A \qquad \Gamma, x{:}A \vdash e \Leftarrow B \hookrightarrow E}{\Gamma \vdash \lambda x.\, e \Leftarrow A \to B \hookrightarrow \lambda x.\, E}\ \text{T-Lam}$$

$$\frac{\Gamma \vdash e \Rightarrow A \hookrightarrow E \qquad A <: B \hookrightarrow E_{sub}}{\Gamma \vdash e \Leftarrow B \hookrightarrow E_{sub}\ E}\ \text{T-Sub}$$

Fig. 2. Wellformedness and type system of F_i.

$$\boxed{\Gamma \vdash A * B}$$

$$\frac{}{\Gamma \vdash \top * A}\ \mathrm{D}\top \qquad \frac{}{\Gamma \vdash A * \top}\ \mathrm{D}\top\mathrm{Sym} \qquad \frac{\alpha * A \in \Gamma \qquad A <: B}{\Gamma \vdash \alpha * B}\ \mathrm{D}\alpha$$

$$\frac{\alpha * A \in \Gamma \qquad A <: B}{\Gamma \vdash B * \alpha}\ \mathrm{D}\alpha\mathrm{Sym} \qquad \frac{\Gamma, \alpha * A_1 \& A_2 \vdash B * C}{\Gamma \vdash \forall(\alpha * A_1).\, B * \forall(\alpha * A_2).\, C}\ \mathrm{D}\forall$$

$$\frac{\Gamma \vdash A_2 * B_2}{\Gamma \vdash A_1 \to A_2 * B_1 \to B_2}\ \mathrm{D}\to \qquad \frac{\Gamma \vdash A_1 * B \qquad \Gamma \vdash A_2 * B}{\Gamma \vdash A_1 \& A_2 * B}\ \mathrm{D}\&\mathrm{L}$$

$$\frac{\Gamma \vdash A * B_1 \qquad \Gamma \vdash A * B_2}{\Gamma \vdash A * B_1 \& B_2}\ \mathrm{D}\&\mathrm{R} \qquad \frac{A *_{ax} B}{\Gamma \vdash A * B}\ \mathrm{DAx}$$

$$\boxed{A *_{ax} B}$$

$$\frac{}{\mathtt{Int} *_{ax} A_1 \to A_2}\ \text{D-Ax}(\mathbb{Z}\to) \qquad \frac{}{\mathtt{Int} *_{ax} \forall(\alpha * B_1).\, B_2}\ \text{D-Ax}(\mathbb{Z}\forall)$$

$$\frac{}{A_1 \to A_2 *_{ax} \forall(\alpha * B_1).\, B_2}\ \text{D-Ax}(\to\forall) \qquad \frac{B *_{ax} A}{A *_{ax} B}\ \text{D-AxSym}$$

Fig. 3. Algorithmic Disjointness.

of rules for determining whether two types are disjoint. After, it will show a few important properties regarding substitution, which will turn out to be crucial to ensure type-safety. Finally, it will discuss the bounds of disjoint quantification and what implications they have on F_i, with a special focus on the $\top$ type and an hypothetical $\bot$ type.

5.1 Algorithmic Rules for Disjointness

The rules for the disjointness judgement are shown in Figure 3, which consists of two judgements.

Main judgement. The judgement $\Gamma \vdash A * B$ says two types A and B are disjoint in a context Γ. The top five rules are novel in relation to the algorithm described in λ_i. D$\top$ and D$\top$Sym say that any type is disjoint to $\top$. This is a major difference with λ_i, where the notion of disjointness explicitly forbids the presence of $\top$ types in intersections. It turns out that even though $\top$ overlaps with every other type, it does not affect coherence in any way. We will come back to this in Section 6. BRUNO: This explanation is too brief! In our previous system top was not disjoint to anything. You need to explain here why is it ok for top to

be disjoint to everything.. Type variables are dealt with two rules: Dα is the base rule; and DαSym is its twin symmetrical rule. Both rules state that a type variable is disjoint to some type A, if Γ contains any subtype of the corresponding disjointness constraint. This rule is a specialization of the more general lemma:

Lemma 4 (Covariance of disjointness).
If $\Gamma \vdash A * B$ *and* $B <: C$*, then* $\Gamma \vdash A * C$.

Proof. By double induction, first on the disjointness derivation and then on the subtyping derivation. The first induction case for Dα does not need the second induction as it is a straightforward application of subtyping transitivity.

The lemma states that if a type A is disjoint to B under Γ, then it is also disjoint to any supertype of B. Note how these two variable rules would allow one to prove $x * x$, for a given type variable x. However, since we are assuming that all contexts are well-formed, it is not possible to make such derivation as x cannot occur free in A. The rule for disjoint quantification D$\forall$ is the last novel rule. It adds a constraint composed of the intersection both constraints into Γ and checks for disjointness in the bodies under that environment. To illustrate this rule, consider the following two types:

$$(\forall(\alpha * \texttt{Int}).\ \texttt{Int}\&\alpha) \qquad (\forall(\alpha * \texttt{Char}).\ \texttt{Char}\&\alpha)$$

The question is under which conditions are those two types disjoint. In the first type α cannot be instantiated with `Int` and in the second case α cannot be instantiated with `Char`. Therefore for both bodies to be disjoint, α cannot be instantiated with either `Int` or `Char`. The rule for disjoint quantification captures this fact by requiring the bodies of disjoint quantification to be checked for disjointness under both constraints. The reader might notice how this intersection does not necessarily need to be well-formed, in the sense that the types that compose it might not be disjoint. The explanation for this underlies in the fact that disjointness is only necessary to guarantee the coherence of elaboration. Introducing arbitrary intersection types in the environment is not problematic, as the disjointness relation does not rely on the target term produced by the subtyping relation. The remaining rules are identical to the original rules, and we will only briefly explain them. The rule for functions D$\rightarrow$ says that two function types are disjoint if and only if their return types are disjoint. The rules dealing with intersection types (D&L and D&R) say that an intersection is disjoint to some type B, whenever both of their components are also disjoint to B. Finally, the rule DAx says two types are considered disjoint if they are judged to be disjoint by the axiom rules, which are explained below.

Axioms. Axiom rules take care of two types with different language constructs. These rules capture the set of rules is that $A *_{ax} B$ holds for all two types of different constructs unless any of them is an intersection type, a type variable, or $\top$. Note that disjointness with type variables is already captured by Dα and DαSym, and disjointness with the $\top$ type is already captured by D$\top$ and D$\top$Sym.

5.2 Stability under Substitution

The combination of polymorphism and disjoint intersection types invalidates various conventional substitution lemmas related to well-formedness and typing. For example, as shown in Section 2, in the type $\forall(A * \texttt{Int}).\,(\texttt{Int}\&A) \to \texttt{Int}$, the type A cannot be substituted by any type. However, under certain conditions, weaker versions of substituion lemmas do hold. The conditions are guaranteed by the type-system by only allowing instantiations of a type variable with types disjoint to the variable's disjointness constraints.

Problematic substitutions. One rule of thumb in disjoint intersection types is that, if a type A is disjoint to a type B, then the intersection $A\&B$ is well-typed. However, during type instantiation (i.e. when type substitution should be stable), both types A and B can change. It should follow naturally that this instantiation will not produce an ill-formed type $A\&B$, or, more generally, disjointness should be stable under substitution. Let us illustrate with an example, showing why disjointness judgements are not invariant with respect to free variable substitution. In other words, why a careless substitution can violate the disjoint constraint in the context. Consider the following judgement, where in the context $\alpha * \texttt{Int}$, α and $\texttt{Int}$ are disjoint:

$$\alpha * \texttt{Int} \vdash \alpha * \texttt{Int}$$

After the substitution of $\texttt{Int}$ for α on the two types, the sentence

$$\alpha * \texttt{Int} \vdash \texttt{Int} * \texttt{Int}$$

is no longer true since $\texttt{Int}$ is clearly not disjoint with itself. This explains the need to ensure that during type-instantiation the target of the substitution is compatible with the disjointness constraint associated with the variable.

Disjoint substitutions. While disjointness cannot be preserved for general substitutions, if appropriate disjointness pre-conditions are met then disjoiness can be preserved. More formally, the following lemma holds:

Lemma 5 (Disjointness is stable under substitution).
If $(x * C) \in \Gamma$ *and* $\Gamma \vdash C * D$*, then* $[x := C]\ \Gamma \vdash [x := C]\ A * [x := C]\ B$*,*

Proof. By induction on the disjointness derivation of C and D. Special attention is needed for the variable case, where it is necessary to prove stability of substitution for the subtyping relation. It is also needed to show that, if C and D do not contain any variable x, then it is safe to make a substitution in the co-domain of the environment.

Well-formedness substitution stability. Typically polymorphic systems with explicit instantiation are required to be shown that their types are stable under substitution, in order to avoid ill-formed types. In the presence of disjoint quantification, we cannot prove such property. However, a weaker version of that property – but strong enough for our type-system's metatheory – can be proven, namely:

Lemma 6 (Types are stable under substitution).
If $\Gamma \vdash A$ *and* $\Gamma \vdash B$ *and* $(\alpha * C) \in \Gamma$ *and* $\Gamma \vdash B * C$*, then* $[\alpha := B]\,\Gamma \vdash [\alpha := B]\,A$.

Proof. By induction on the well-formedness derivation of A. The intersection case requires the use of Lemma 5. Also, the variable case required proving that if α does not occur free in A, and it is safe to substitute it in the co-domain of Γ, then it is safe to perform the substitution.

This lemma enables us to show that all types produced by the type-system are well-typed. More formally, we have that:

Lemma 7 (Well-formed typing).
If $\Gamma \vdash e \Leftarrow A$*, then* $\Gamma \vdash A$.
If $\Gamma \vdash e \Rightarrow A$*, then* $\Gamma \vdash A$.

Proof. By induction on the derivation and applying Lemma 6 in the case of T-TApp.

Even though the meta-theory is consistent with the expected results, there is still a question that remains unanswered: what exactly are the bounds of disjoint quantification? In other words, which type(s) might be used to allow unrestricted instantiation, and which one(s) might be used to completely restrict instantiation? As one might expect, the answer is tightely related to subtyping, as we will show next.

5.3 Bounds of Disjoint Quantification

Substitution raises the question of what range of types can be instantiated for a given variable, under a given context. To get a feeling about this, we ask the reader to recall Lemma 4, which was used to justify the rule for disjointness of variables. If one takes A as some variable α, then the lemma should read as:

$$\frac{\Gamma \vdash \alpha * B \qquad B <: C}{\Gamma \vdash \alpha * C}$$

Now we can ask: how many suitable types are there to instantiate α with? Before we answer this, let us ask first how many options are there for C, depending on the shape of B? Given that the cardinality of F_i's types is infinite, for the sake of this example we will restrict the type universe to a finite number of primitive types (i.e. `Int`, `String`, etc), disjoint intersections of these types, $\top$ and $\bot$. Having this in mind, we can answer the second question: the number of choices for C is directly proportional to the number of intersections present in B. For example, taking B as `Int` leads C to be either $\top$ or `Int`; whereas B as `Int&String` leaves C as either $\top$, `Int` or `String`. However, as the choices for C grows, the less choices we are left to instantiate the variable α, since α must be disjoint to all possible C's. Thus, to answer the first question, the options for instantiating α are inversely proportional to the number of intersections present

in B. As an analogy, one might think of a disjointness constraint as a set of (forbidden) types, where primitive types are the singleton set and each & is the set union.

We can now turn our attention to the two extreme cases, namely $\top$ (i.e. the 0-ary intersection) and $\bot$ (i.e. the infinite intersection) [5]. Following the same logic, we may conclude that $\top$ as the associated constraint leaves α with the most options for instantiation whereas $\bot$ will deliver the least options. This also implies that $\top$ is the empty constraint, meaning that a variable associated to it can be instantiated to any well-formed type. It is a subtle but very important property, since $\mathsf{F_i}$ is a generalization of System F. This means that any type variable present in a type signature of System F, can be represented in $\mathsf{F_i}$ equivalently by assigning it a $\top$ disjointness constraint (as seen in Section 2.4).

6 Semantics, Coherence and Type-Safety

This section discusses the elaboration semantics of $\mathsf{F_i}$ and proves type-safety and coherence.

6.1 Semantics

The dynamic semantics of the call-by-value $\mathsf{F_i}$ is defined by means of a type-directed translation to an extension of System F with pairs.

Target language. The syntax and typing of our target language is unsurprising:

$$
\begin{array}{lll}
\text{Types} & \mathsf{T} ::= \alpha \mid \mathtt{Int} \mid \mathsf{T_1} \to \mathsf{T_2} \mid \forall \alpha.\, \mathsf{T} \mid \mathsf{()} \mid \mathsf{(T_1, T_2)} \\
\text{Terms} & \mathsf{E} ::= x \mid i \mid \lambda x.\, \mathsf{E} \mid \mathsf{E_1\ E_2} \mid \Lambda \alpha.\, \mathsf{E} \mid \mathsf{E\ T} \mid \mathsf{()} \mid \mathsf{(E_1, E_2)} \mid \mathtt{proj_1}\,\mathsf{E} \mid \mathtt{proj_2}\,\mathsf{E} \\
\text{Contexts} & \mathsf{G} ::= \cdot \mid \mathsf{G}, \alpha \mid \mathsf{G}, x{:}\mathsf{T}
\end{array}
$$

The highlighted part shows its difference with the standard System F. The interested reader can find the formalization of the target language syntax and typing rules (which are standard) in our Coq development.

Type and context translation. Figure 4 defines the type translation function $|\cdot|$ from $\mathsf{F_i}$ types A to target language types T. The notation $|\cdot|$ is also overloaded for context translation from $\mathsf{F_i}$ contexts Γ to target language contexts G.

BRUNO: This figure is too wastefull in terms of syntax. Use a table or tabular to put the two translations side-by-side. The Figure should take half of the space that it currently takes.

6.2 Top-like types and their coercions

BRUNO: Save space in the figure, by putting the definitions side-by-side.

[5] $\bot$ would not add anything to the hypothetical finite type system, however it can be seen as the infinite intersection in $\mathsf{F_i}$.

$\boxed{|A| = T}$

$$
\begin{aligned}
|\alpha| &= \alpha \\
|\top| &= () \\
|A_1 \to A_2| &= |A_1| \to |A_2| \\
|\forall(\alpha * A).\, B| &= \forall\alpha.\, |B| \\
|A_1 \& A_2| &= (|A_1|, |A_2|)
\end{aligned}
$$

$\boxed{|\Gamma| = G}$

$$
\begin{aligned}
|\cdot| &= \cdot \\
|\Gamma, \alpha * A| &= |\Gamma|, \alpha \\
|\Gamma, \alpha{:}A| &= |\Gamma|, \alpha{:}|A|
\end{aligned}
$$

Fig. 4. Type and context translation.

$\boxed{\rceil A \lceil}$

$$
\frac{}{\rceil \top \lceil}\ \mathrm{TL}\top \qquad \frac{\rceil A \lceil \quad \rceil B \lceil}{\rceil A \& B \lceil}\ \mathrm{TL}\& \qquad \frac{\rceil B \lceil}{\rceil A \to B \lceil}\ \mathrm{TL}\to \qquad \frac{\rceil A \lceil}{\rceil \forall(\alpha * B).\, A \lceil}\ \mathrm{TL}\forall
$$

$\boxed{[\![A]\!]_C = T}$

$$
[\![A]\!]_C = \begin{cases} \rceil A \lceil & [\![A]\!] \\ \text{otherwise} & C \end{cases}
$$

$\boxed{[\![A]\!] = T}$

$$
[\![A]\!] = \begin{cases} A = \top & () \\ A = A_1 \to A_2 & \lambda x.\, [\![A_2]\!] \\ A = A_1 \& A_2 & ([\![A_1]\!], [\![A_2]\!]) \\ A = \forall(\alpha * B).\, A & \Lambda\alpha.\, [\![A]\!] \end{cases}
$$

Fig. 5. Top-like types and their coercions.

Our definition of top-like types is naturally extended from λ_i. BRUNO: What are top-like types and why are they important? The reader will not know this, so you need to spend some sentences saying something about it.

JOAO: place this sentence somewhere here This is due to top-like types only having one inhabitant each and their (unique) coercions can be directly generated from the structure of their type.

The rules that compose this unary relation, denoted as $\rceil.\lceil$, are presented at the top of Figure 5. The only new rule is $\mathsf{TL}\forall$, which extends the notion of top-like types for the (disjoint) universal quantifier.

It is important pointing out that, despite the small extension in the definition of top-likes types, the notion of disjointness has also changed, leading to a larger set of well-formed top-like types. Thus, to retain coherece, we adjusted the meta-function $[\![A]\!]$, as shown in the bottom of Figure 5. Note how not only the $\forall$ case is now defined, but also the intersection case (covering types such as $\top\&\top$). These changes are very important since they play a fundamental role in ensuring the coherence of subtyping for top-like types.

6.3 Coercive Subtyping and Coherence

Coercive subtyping. The judgement

$$A_1 <: A_2 \hookrightarrow E$$

extends the subtyping judgement in Figure 1 with a coercion on the right hand side of $\hookrightarrow$. A coercion E is just a term in the target language and is ensured to have type $|A_1| \to |A_2|$ (by Lemma 8). For example,

$$\mathtt{Int}\&\alpha <: \alpha \hookrightarrow \lambda x.\, \mathtt{proj}_2\, x$$

generates a target coercion function with type: $(\mathtt{Int}, \alpha) \to \alpha$.

In rule S$\top$ the coercion is the constant function of the unit term. In rules Sα, S$\mathbb{Z}$, coercions are just identity functions. In S$\to$, we elaborate the subtyping of parameter and return types by η-expanding f to $\lambda x.\, f\, x$, applying E_1 to the argument and E_2 to the result. Rules S&L$_1$, S&L$_2$, and S&R elaborate intersection types. S&R uses both coercions to form a pair. Rules S&L$_1$ and S&L$_2$ reuse the coercion from the premises and create new ones that cater to the changes of the argument type in the conclusions. Rule S$\forall$ elaborates disjoint quantification, reusing only the coercion of subtyping between the bodies of both types. Rule SRec elaborates records by simply reusing the coercion generated between the inner types. Finally, all rules produce type-correct coercions:

Lemma 8 (Subtyping rules produce type-correct coercions). *If* $A_1 <: A_2 \hookrightarrow E$, *then* $\cdot \vdash E : |A_1| \to |A_2|$.

Proof. By a straightforward induction on the derivation.

Unique coercions In order to ensure a coherent type-system, the subtyping relation also needs to be coherent. With disjoint polymorphism the following theorem holds:

Lemma 9 (Unique subtype contributor).
If $A_1 \& A_2 <: B$*, where* $A_1 \& A_2$ *and* B *are well-formed types, and* B *is not top-like, then it is not possible that the following holds at the same time:*

1. $A_1 <: B$
2. $A_2 <: B$

Proof. By double induction: the first on the disjointness derivation (which follows from $A_1 \& A_2$ being well-formed); the second on type B. The variable cases $D\alpha$ and $D\alpha Sym$ needed to show that, for any two well-formed and disjoint types A and B, and B is not toplike, then A cannot be a subtype of B.

Using the unique subtype contributor, we can show that the coercion of a subtyping relation $A <: B$ is uniquely determined. This fact is captured by the following lemma:

Lemma 10 (Unique coercion).
If $A <: B \hookrightarrow E_1$ *and* $A <: B \hookrightarrow E_2$*, where* A *and* B *are well-formed types, then* $E_1 \equiv E_2$.

Proof. By induction on the first derivation and case analysis JOAO: or say inversion instead? on the shape of the second.

6.4 Elaboration of type-system and coherence

In order to prove the coherence result, we refer to the previously introduced bidirectional type-systemBRUNO: Just use a reference to the appropriate (sub)section here: *never* use vague references. The bidirectional type-system is elaborating, producing a term in the target language while performing the typing derivation.

Key idea of the translation. This translation turns merges into usual pairs, similar to Dunfield's elaboration approach [13]. It also translates the form of disjoint quantification and disjoint type application into regular (polymorphic) quantification and type application. For example,

$$\Lambda(\alpha * \texttt{Int}).\, \lambda x.\, (x, , 1)$$

in F_i will be translated into System F's:

$$\Lambda\alpha.\, \lambda x.\, (x, 1)$$

The translation judgement. The translation judgement $\Gamma \vdash e : A \hookrightarrow E$ extends the typing judgement with an elaborated term on the right hand side of $\hookrightarrow$. The translation ensures that E has type $|A|$. We discuss the most relevant rules/coercions in greater detail next. The two new rules for type abstraction (T-BLam) and type application (T-TApp) generate the expected corresponding coercions in System F. In F_i, subtyping means, for example, that one may pass more information to a function than what is required. This is not the case

in System F. The rule T-Sub is used to account for such differences: the coercion $\mathsf{E}_{\mathtt{sub}}$, derived from the subtyping relation, is applied to coerce the System F term into the right type. The coercions generated for T-Rec and T-ProjR simply erase the labels (since there are no labels in the target language) and translate the corresponding underlying term. It is also noteworthy pointing out that, as usual, the rule T-Merge translates merges into pairs.

Type-safety The type-directed translation is type-safe. This property is captured by the following two theorems.

Theorem 1 (Type preservation). *We have that:*

- *If* $\Gamma \vdash e \Rightarrow A \hookrightarrow E$, *then* $|\Gamma| \vdash E : |A|$.
- *If* $\Gamma \vdash e \Leftarrow A \hookrightarrow E$, *then* $|\Gamma| \vdash E : |A|$.

Proof. By structural induction on the term and the corresponding inference rule.

Theorem 2 (Type safety). *If* e *is a well-typed* F_{i} *term, then* e *evaluates to some System* F *value* v.

Proof. Since we define the dynamic semantics of F_{i} in terms of the composition of the type-directed translation and the dynamic semantics of System F, type safety follows immediately.

Uniqueness of type-inference An important property of the bidirectional type-checking is that, given an expression e, if it is possible to infer a type for it, then e has a unique type.

Theorem 3 (Uniqueness of type-inference). *We have that:*

- *If* $\Gamma \vdash e \Rightarrow A_1 \hookrightarrow E_1$ *and* $\Gamma \vdash e \Rightarrow A_2 \hookrightarrow E_2$, *then* $A_1 = A_2$.

Proof. By structural induction on the term and the corresponding inference rule.

Coherency of Elaboration Combining the previous results, we are finally able to show the central theorem:

Theorem 4 (Unique elaboration). *We have that:*

- *If* $\Gamma \vdash e \Rightarrow A \hookrightarrow E_1$ *and* $\Gamma \vdash e \Rightarrow A \hookrightarrow E_2$, *then* $E_1 \equiv E_2$.
- *If* $\Gamma \vdash e \Leftarrow A \hookrightarrow E_1$ *and* $\Gamma \vdash e \Leftarrow A \hookrightarrow E_2$, *then* $E_1 \equiv E_2$.

("$\equiv$" means syntactical equality, up to α-equality.)

Proof. By induction on the first derivation. The most important case is the subsumption rule:

$$\frac{\Gamma \vdash e \Rightarrow A \hookrightarrow E \qquad A <: B \hookrightarrow E_{\mathtt{sub}}}{\Gamma \vdash e \Leftarrow B \hookrightarrow E_{\mathtt{sub}}\ E}\ \text{T-Sub}$$

We need to show that E_{sub} is unique (by Lemma 10), and thus to show that A is well-formed (by Lemma 7). Note that this is the place where stability of substitutions (used by Lemma 7) plays a crutial role in guaranteeing coherence. We also need to show that A is unique (by Theorem 3). Uniqueness of A is needed to apply the induction hypothesis.

7 Related Work

BRUNO: This section needs some major rewriting. Remeber that the goal of writting the related work section is to identify how previous work is different from ours. The text written here is also slightly outdated compared to what we wrote on ICFP. In particular we discuss the work by Castagna on merges at ICFP.

Intersection types, polymorphism and the merge operator. To our knowlegde no previous work has been done in a caluclus which includes parametric polymorphism, intersection types and a *merge* operator.

Concerning simply-typed systems, the oldest work dates back to the 1980s, when Reynolds invented Forsythe [26], with his merge-like operator being noted as $\mathtt{p}_1, \mathtt{p}_2$. Even though his calculus was proven to be coherent [25], it uses a form of biased choice by having precedence in typing rules for intersections. His system is also arguably more restrictive than ours since it forbids, for instance, the merge of two functions.

Pierce [21] made a comprehensive review of coherence, but he was unable to prove it for his $\mathsf{F}_{\wedge}$ calculus. He introduced a primitive `glue` function as a language extension which corresponds to our merge operator. However, users can "glue" two arbitrary values, which may lead to incoherence.

More recently, Dunfield [13] formalised a system with intersection types and a merge operator with a type-directed translation to the simply-typed lambda calculus with pairs. The major differences to our calculus (besides being simply-typed), is that on one hand his system includes union types, but on the other it lacks coherence.

Our work is an extension to the recent λ_{i} [27]. F_{i} solves the coherence problem present in Dunfield's calculus, by requiring that intersection types can only be composed of *disjoint* types. λ_{i} uses a specification for disjointness, that reads as:

JOAO: put definition here

F_{i} does not use such definition as its adaptation to polymorphic types would require using some form of unification. Since the purpose of such specification is its simplicity and readability (compared to the algorithmic counterpart), we argue that unification would go against this principle, and opted to drop the specification. In fact, F_{i}'s notion of disjointness is based on λ_{i}'s definition of the more general notion of disjointess concerning $\top$ types, called $\top$-disjointness:

JOAO: put definition here

A major difference between the two calculus, is that $\top$-disjointness does not allow $\top$ in intersections, while our calculus allows it. As a consequence, the set

of well-formed *top-like* types is a superset of F_{i}'s. The reason for this is covered in greater detail in Section **??** JOAO: put ref to section.

JOAO: reference other calculus

Intersection types and polymorphism, without the merge operator.

Extensible records. GEORGE: Record field deletion is also possible.

Encoding records using intersection types appeared in Reynolds [26] and Castagna et al. [10]. Although Dunfield also discussed this idea in his paper [13], he only provided an implementation but not a formalization. Very similar to our treatment of elaborating records is Cardelli's work [6] on translating a calculus, named $\mathsf{F}_{<:\rho}$, with extensible records to a simpler calculus that without records primitives (in which case is $\mathsf{F}_{<:}$). But he did not consider encoding multi-field records as intersections; hence his translation is more heavyweight. Crary [12] used intersection types and existential types to address the problem that arises when interpreting method dispatch as self-application. But in his paper, intersection types are not used to encode multi-field records.

Wand [28] started the work on extensible records and proposed row types [29] for records. Cardelli and Mitchell [8] defined three primitive operations on records that became standard in type-systems with record types: *selection*, *restriction*, and *extension*. Examples of calculus that use such primitives are ... Following Cardelli and Mitchell's approach, which is based on extension and row types, both Leijen's systems [17,18] and ours allow records that contain duplicate labels. Leijen's system is more sophisticated, since it relies on shadowing of labels. Also, it supports passing record labels as arguments to functions. He also showed an encoding of intersection types using first-class labels.

Later ... defined a fourth operation on records called *concatenation*. The operation eventually became standard of many systems, even though its semantics can be usually defined in terms of extension. One problem with concatenation is that tipically systems are hard to use in practise, due to the complicated set of constraints/filters which might arise from combining several, possibly polymorphic, records [18]. The merge operator in F_{i} plays the same role as concatenation, as its components may contain any (well-formed and disjoint) types. However.. JOAO: type signatures are simpler and piggybacking on more general notion(s) of DIT

Traits and Mixins

8 Conclusion and Future Work

This paper described $\mathsf{F}^{*}_{\mathsf{i}}$: a System F-based language that combines intersection types, parametric polymorphism and a merge operator. The language is proved to be type-safe and coherent. To ensure coherence the type system accepts only disjoint intersections. To provide flexibility in the presence of parametric polymorphism, universal quantification is extended with disjointness constraints. We

believe that disjoint intersection types and disjoint quantification are intuitive, and at the same time expressive.

We implemented the core functionalities of the F_i^* as part of a JVM-based compiler. Based on the type system of F_i^*, we have built an ML-like source language compiler that offers interoperability with Java (such as object creation and method calls). The source language is loosely based on the more general System F_ω and supports a number of other features, including records, mutually recursive `let` bindings, type aliases, algebraic data types, pattern matching, and first-class modules that are encoded using `letrec` and records.

For the future, we intend to improve our source language and show the power of disjoint intersection types and disjoint quantification in large case studies. We are also interested in extending our work to systems with a $\top$ type. This will also require an adjustment to the notion of disjoint types. A suitable notion of disjointness between two types A and B in the presence of $\top$ would be to require that the only common supertype of A and B is $\top$. Finally we would like to study the addition of union types. This will also require changes in our notion of disjointness, since with union types there always exists a type $A|B$, which is the common supertype of two types A and B.

Bottom BRUNO: I think the discussion about bottom can be done later, perhaps in future work. JOAO: rephrase this Inversely, the most restrictive type is $\bot$, as it is not disjoint to any type, except top-like types. JOAO: is this true? I'm not sure about what (bot & top) means. However, introducing $\bot$ is not compatible with our disjointness rule $D\alpha$ and well-formedness of contexts. Let us take a closer look, by supposing that we wish to derive $\Gamma \vdash x * x$, for some variable x, under some well-formed context Γ. In F_i, we can only use $D\alpha$ with the type A as a sub-type of x, i.e. an (n-ary) intersection containing x. Well-formedness of environments guarantees that this will never happen, since x is not in scope of itself. Thus, without a $\bot$ type, a derivation for that statement does not exist. However, by introducing $\bot$ we may now construct such derivation, as A can now be $\bot$: a valid sub-type of x which does not contain x. In fact, had F_i included a $\bot$ type, then introducing any *bottom-like* type (i.e. $\bot \& A$, for any type A) can lead to this undesired behaviour. Since defining the lower bound is not strictly necessary to the formalization; introduces substantial complexity in our system; and its practical application is not clear, we left this as an open problem for future work.

References

1. Ceylon, https://ceylon-lang.org/
2. The Coq Proof Assistant, https://coq.inria.fr/
3. Flow, https://flowtype.org/
4. TypeScript, https://www.typescriptlang.org/
5. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Necula, G.C., Wadler, P. (eds.) Proceeding of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 3–15. ACM (2008)
6. Cardelli, L.: Extensible records in a pure calculus of subtyping. Digital. Systems Research Center (1992)
7. Cardelli, L., Martini, S., Mitchell, J.C., Scedrov, A.: An extension of system f with subtyping. Inf. Comput. 109(1-2) (Feb 1994)
8. Cardelli, L., Mitchell, J.C.: Operations on records. In: Mathematical foundations of programming semantics (1990)
9. Castagna, G., Ghelli, G., Longo, G.: A calculus for overloaded functions with subtyping. In: Proceedings of the 1992 ACM Conference on LISP and Functional Programming. LFP '92 (1992)
10. Castagna, G., Ghelli, G., Longo, G.: A calculus for overloaded functions with subtyping. Information and Computation (1995)
11. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional characters of solvable terms. Mathematical Logic Quarterly 27(2–6), 45–58 (1981)
12. Crary, K.: Simple, efficient object encoding using intersection types. Tech. rep., Cornell University (1998)
13. Dunfield, J.: Elaborating intersection and union types. Journal of Functional Programming (2014)
14. Harper, R., Pierce, B.: A record calculus based on symmetric concatenation. In: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM (1991)
15. Harper, R.W., Pierce, B.C.: Extensible records without subsumption (1990)
16. Jones, M., Jones, S.: Lightweight extensible records for haskell (1999)
17. Leijen, D.: First-class labels for extensible rows. UU-CS (2004-051) (2004)
18. Leijen, D.: Extensible records with scoped labels. Trends in Functional Programming (2005)
19. Odersky, M., et al.: An overview of the Scala programming language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland (2004)
20. Oliveira, B.C.d.S., Van Der Storm, T., Loh, A., Cook, W.R.: Feature-oriented programming with object algebras. In: ECOOP 2013–Object-Oriented Programming (2013)
21. Pierce, B.C.: Programming with Intersection Types and Bounded Polymorphism. Ph.D. thesis, Carnegie Mellon University (December 1991)
22. Pierce, B.C., Turner, D.N.: Simple type-theoretic foundations for object-oriented programming. J. Funct. Program. 4(2), 207–247 (1994)
23. Pottinger, G.: A type assignment for the strongly normalizable λ-terms. In: To H. B. Curry: essays on combinatory logic, lambda calculus and formalism. pp. 561–577. Academic Press, London (1980)
24. Rémy, D.: Type inference for records in a natural extension of ML. Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design. MIT Press (1993)

25. Reynolds, J.C.: The coherence of languages with intersection types. In: Proceedings of the International Conference on Theoretical Aspects of Computer Software. TACS '91 (1991)
26. Reynolds, J.C.: Design of the Programming Language Forsythe, pp. 173–233. Birkhäuser Boston, Boston, MA (1997)
27. d. S. Oliveira, B.C., Shi, Z., Alpuim, J.: Disjoint intersection types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016. pp. 364–377 (2016)
28. Wand, M.: Complete type inference for simple objects. In: LICS (1987)
29. Wand, M.: Type inference for record concatenation and multiple inheritance. In: Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on. IEEE (1989)