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 core 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, while ensuring coherence 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, that 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 disjoint intersection types, while retaining coherence.

1 Introduction

Intersection types are an incresingly popular language feature for many modern programming languages. Examples of languages that include some form of intersection types are Microsoft's TypeScript [], Redhat's Ceylon [], Facebook's Flow [] and Scala []¹[1]. In those languages a typical use of intersection types, which has been known for a long time [], is to model the subtyping aspects of some (possibly restricted) form 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 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
```

---

[1] Limited forms of intersection types also exist in recent versions of Java [].

which given an argument `x` with an intersection type `B with C` weaken the type to `B`. In Java, which does not support first-class intersections, such method cannot be directly expressed.

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[2] 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!)*

There are two points worth emphasizing here. 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. In typical statically typed class-based languages composition is done at compile-time via inheritance (and the type of composed objects has to be statically known). In contrast, in prototype-based languages objects are composed at run-time, and their types are not necessarally (fully) 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 not captured by 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 idea is that the new object can answer to method calls that could be done to either `first` and `second`. 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 [?,?,18,?] as an introduction form for intersection types. As we have seen the merge operator is encodable in TypeScript using low-level features. Similar encodings have been proposed for Scala to enable applications where the merge operator plays a fundamental role []. Unfortunatelly, the merge operator is not directly supported by TypeScript, Scala, Ceylon or Flow. A possible reason for this is that most calculi with intersection types in the literature lack the merge operator. While powerful, the merge operator is well-known to introduce *(in)coherence* problems []. If care is not taken,

---

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

certain programs using the merge operator do not have a unique semantics, which significantly complicates reasoning about programs.

Recent work proposed using *disjoint intersection 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, which guarantees coherence. However, such work was done in a simply typed setting.

Combining parametric polymorphism wiht 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 try 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 would prevent many useful programs, including the `extend` function.

This paper presents F_i: a core calculus with a variant of *intersection types*, a form of *parametric polymorphism* and a *merge operator*. The key innovation in the calculus is *disjoint polymorphism*: a constrained form of parametric polymorphism, that 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 our calculus the `extend` function trivially implemented as follows:

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

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 System 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.
- **Applications:** We show that in F_i^* type-theoretic encodings can be combined with subtyping to provide extensibility.
- **Implementation:** An implementation of an extension of System F_i^*, as well as the examples presented in the paper, are publicly available[3].

2 Overview

JOAO: review this paragraph once the section is finished This section introduces F_i and its support for intersection types, parametric polymorphism and the

---

[3] **Note to reviewers:** Due to the anonymous submission process, the code (and some machine checked proofs) is submitted as supplementary material.

merge operator. It then discusses the issue of coherence and shows how the notion of disjoint intersection types and disjoint quantification achieve a coherent semantics.

Note that 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, our implementation supports them.

2.1 Intersection Types and the Merge Operator

Intersection types date back as early as Coppo et al.'s work [11]. Since then various researchers have studied intersection types, and some languages have adopted them in one form or another.

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 either values of type A or of type B. For instance, consider the following program in F_i:

```
let x : Int & Char = ... in -- definition omitted
let idInt (y : Int) : Int = y in
let idChar (y : Char) : Char = y in
(idInt x, idChar x)
```

BRUNO: I had comment about not having the same text. I should have perhaps have been a bit more clear and say that having the same code is ok, it is just the text that needs a little rewriting. In fact now the code examples are worst: identity functions are pretty boring and they don't really illustrate the point.

If a value `x` has type `Int & Char` then `x` can be used anywhere where either values of type `Int` or a `Char` are expected. This means that, in the program above the functions `idInt` and `idChar` – the identity functions on integers and characters, 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*. JOAO: add citations? As Dunfield [18] 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` is constructed in F_i as follows:

```
let x : Int & Char = 1,,'c' in ...
```

In F_i (following Dunfield's notation), the merge of two values v_1 and v_2 is denoted as $v_1,,v_2$.

Merge vs Pairs The significant difference between intersection types with a merge operator and regular pairs is in the elimination construct. With pairs there are explicit eliminators (`fst` and `snd`), and these eliminators must be used to extract the components of the right type. With intersection types and a merge

operator, elminators are implicit in the language, meaning no uses of projection functions are necessary.

2.2 Coherence and Disjointness

Coherence is a desirable property for a semantics. A semantics is said to be coherent if any *valid program* has exactly one meaning [30] (that is, the semantics is not ambiguous). Unfortunately the implicit nature of elimination for intersection types built with a merge operator can lead to incoherence, This is due to intersections with overlapping types, as in `Int&Int`. For example, the result of the program (`(1,,2) : Int`) can be either `1` or `2`, depending on the implementation of the language.

One option to restore coherence is to reject programs which may have multiple meanings. The λ_i JOAO: add reference system – a calculus with a simply-typed calculus with intersection types and a merge operator, in the same flavour of Dunfield's – solves this problem by using the concept of disjoint intersections.

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.

This is precisely the approach taken in λ_i: a merge can only be composed of two values as long as their types are *disjoint*. Disjointness is a binary relation between two types, defined for any types which do not contain any overlapping types. It can be specified as: given any two types, they are disjoint if there does not exist any common super-type.

With this concept of disjointness in mind, it is easy to verify that the previous example will no longer be accepted since (`(1,,2) : Int`) is no longer well-typed! The merge operator requires two types to be disjoint in order to type-check it into an intersection. In the example (`1,,2`) is rejected by the compiler, since `Int` is not disjoint with `Int`. In other words, there exists a super-type of both `Int` and `Int`, namely `Int` itself. This result can be generalized and λ_i has shown that it can lead to a coherent calculus. Although this is a promising result, the question remains: is it possible to incorporate parametric polymorphism in such calculus, while retaining coherence?

2.3 Parametric Polymorphism

Unfortunately, adding parametric polymorphism is non-trivial. A naive attempt to add polymorphism would consist in introducing a forall type, type variables, and a big lambda at term level. Type variables can be assumed as disjoint to any other type, as a starting point. Now consider the attempt to write the following polymorphic function in such system (we will use uppercase Latin letters to denote type variables):

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

The `fst` function is supposed to extract a value of type `A` from the merge value `x` (of type `A&B`). This function is problematic: when `A` and `B` are instantiated to non-disjoint types, then uses of `fst` may lead to incoherence. For example, consider the following use of `fst`:

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

This program is clearly incoherent as both 1 and 2 can be extracted from the merge and still match the type of the first argument of `fst`.

Biased choice breaks equational reasoning. At first sight, one option to workaround the incoherence issue would be to bias the type-based merge lookup to the left or to the right. However, biased choice is very problematic when parametric polymorphism is present in the language. To see the issue, suppose we chose to always pick the rightmost value in a merge when multiple values of same type exist. Intuitively, it would appear that the result of the use of `fst` above is 2. Indeed simple equational reasoning seems to validate such result:

```
    fst Int Int (1,,2)
↝ ((fun z → z) : Int → Int) (1,,2)  --  By the definition of fst
↝ ((fun z → z) : Int → Int) 2        --  Right-biased coercion
↝ 2                                   --   By β-reduction
```

However (assuming a straightforward implementation of right-biased choice) the result of the program would be 1! The reason for this has todo with *when* the type-based lookup on the merge happens. In the case of `fst`, lookup is triggered by a coercion function inserted in the definition of `fst` at compile-time. In the definition of `fst` all it is known is that a value of type A should be returned from a merge with an intersection type A&B. Clearly the only type-safe choice to coerce the value of type A&B into A is to take the left component of the merge. This works perfectly for merges such as `(1,,'c')`, where the types of the first and second components of the merge are disjoint. For the merge `(1,,'c')`, if a integer lookup is needed, then `1` is the rightmost integer, which is consistent with the biased choice. Unfortunately, when given the merge `(1,,2)` the left-component `1` is also picked up, even though in this case `2` is the rightmost integer in the merge.

The subtle interaction of polymorphism and type-based lookup means that equational reasoning is broken. In the equational reasoning steps above, doing apparently correct substitutions lead us to a wrong result. This is a major problem for biased choice and a reason to dismiss it as a possible implementation choice for F_i.

A more conservative attempt. Another attempt at restoring coherence can be to forbid type variables inside intersections (i.e. type variables are not disjoint to any type). This conservative approach would solve the problem of coherence, but it would also greatly restrict the expressiveness of the resulting language. For example, the function `fst` defined above, would no longer be accepted by

the system. In fact, parametric polymorphism and intersection types could only be mixed in a very limited manner - as long as variables do not reside under intersections - and this is arguably a useful improvement in respect to other standard type systems, such as System F.

2.4 Disjoint Quantification

To avoid being overly conservative, while still retaining coherence in the presence of parametric polymorphism and intersection types, F_i uses an extension to universal quantification called *disjoint quantification*. Inspired by bounded quantification [5], where a type variable is constrained by a type bound, disjoint quantification allows a type variable to be constrained so that it is disjoint with a given type. With disjoint quantification a variant of the program fst, which is accepted by F_i, would be written as:

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

The small change is in the declaration of the type parameter B. The notation $B * A$ means that in this program the type variable B is constrained so that it can only be instantiated with any type disjoint to A. This ensures that the merge denoted by x 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 are rejected. However, the following use of fst:

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

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

Empty constraint Even though disjoint quantification solves the problem of coherence, there is still one detail that needs further justification. The reader might have noticed how we ommitted the disjointness constraint of the type variable A in the fst function. 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 with all types except the intersection 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 $A * \top$. 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 $\forall(A * \top).A \to A$.

2.5   Stability of Substitutions

From the technical point of view, the main challenge in the design of System 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 holds: BRUNO: state usual property here

In F_i if a type variable A is substituted in a type T_1, for a type T_2 (written $[A := T_2] \, T_1$), 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 $\forall(B * \text{Int}).(\text{Int}\&B) \to \text{Int}$. Now, the second `Int` application is problematic, since instantiating B with `Int` will lead to the ill-formed type $(\text{Int}\&\text{Int}) \to \text{Int}$. However, from this example it is easy to see that all types which are not problematic are exactly the 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 F_i we show two applications. The first application shows how to mimic TypeScript's documentation example of dynamic mixins in F_i. The second application shows how F_i enables a powerful form of polymorphic extensible records. BRUNO: I think we should swap the order of the examples: TypeScript mixins first, and extensible records second.

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

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 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. BRUNO: Why do w have two versions of the code in this section? Can we remove one.

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:

$$\text{select} :: \forall (\alpha * \top). \{l : \alpha\} \to \alpha$$
$$\text{select} = \Lambda(\alpha * \top). \lambda x. x.l$$

```
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 recorld $\{l : \alpha\}$. This resembles other systems with subtyping ..., although it is slightly more general, as any it is not restricted only to record types. JOAO: references

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:

$$\text{remove} :: \forall (\alpha * \top). \forall (r * \{l : \alpha\}). (\{l : \alpha\} \& r) \to r$$
$$\text{remove} = \Lambda(\alpha * \top). \Lambda(r * \{l : \alpha\}). \lambda x. x$$

```
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, (Harper & Pierce) 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, a function which concatenates a single record with field l of type `Int` with another record that lacks this field, is the following (slightly modified in terms of notation):

$$\text{addL}_1 :: \forall \alpha \# l. \alpha \to (\alpha \| \{l : \texttt{Int}\})$$
$$\text{addL}_1 = ...$$

```
let addL₁ (A # l) (x : A) : (a || { l : Int }) = ...
in ...
```

The reader might notice the resemblance with our system:

$$\text{addL}_2 :: \forall (\alpha * \{l : \texttt{Int}\}). \alpha \to (\alpha \& \{l : \texttt{Int}\})$$
$$\text{addL}_2 = ...$$

```
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 what (Harper & Pierce) describe in their system. 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:

$$\text{addL}_3 :: \forall (\beta * \top). \forall (\alpha * \{l : \beta\}). \alpha \to (\alpha \& \{l : \beta\})$$
$$\text{addL}_3 = ...$$

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

Unfortunately, our system does not support the ⊥ 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 rises the question: how would one classify our system in terms of extension? As noted in [22], 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 ... (SPJ & MJ) – a strict system with extension – an example of a function that uses record types is the following:

$$\text{average}_1 :: (r\backslash y, r\backslash x) \Rightarrow \{r|x :: \texttt{Int}, y :: \texttt{Int}\} \to \texttt{Int}$$
$$\text{average}_1 \; r = (r.x + r.y)/2$$

```
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 [23], the more general program would be accepted:

$$\text{average}_2 :: \forall \; r \; x \; y, \{x :: \texttt{Int}, y :: \texttt{Int}|r\} \to \texttt{Int}$$
$$\text{average}_2 \; r = (r.x + r.y)/2$$

```
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 $\mathsf{F_i}$, this program could be re-written as [4]:

$$\text{average}_3 :: \forall(r * \{x : \texttt{Int}\} \& \{y : \texttt{Int}\}).\{x : \texttt{Int}\} \& \{y : \texttt{Int}\} \& r \to \texttt{Int}$$
$$\text{average}_3 = \Lambda(t * \{x : \texttt{Int}\} \& \{y : \texttt{Int}\}).\lambda r.\, (r.x + r.y)/2$$

```
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 $\mathsf{F_i}$ 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?) $\mathsf{F_i}$ does not suffer from this problem. JOAO: since we have no refinement types? We may illustrate by defining a suitable update function, in a similar fashion to [23]:

$$\text{update} :: \forall(\alpha * \top).\forall(r * \{l : \alpha\}).\{l : \alpha\} \& r \to \beta \to \{l : \beta\} \& r$$
$$\text{update} = \Lambda(\alpha * \top).\Lambda(r * \{l : \alpha\}).\lambda x.\lambda v.\{l = v\},, (\text{remove } \alpha \text{ } r \text{ } x)$$

```
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 $\mathsf{F_i}$ Calculus

This section presents the syntax, subtyping, and typing of $\mathsf{F_i}$: a calculus with intersection types, parametric polymorphism, records and a merge operator. This calculus is an extension of λ_i and Dunfield's calculus [18], which are simply typed calculus with intersection types and a merge operator. The novelty of $\mathsf{F_i}$ 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 $\mathsf{F_i}$ has been mechanized in Coq, which is available in the supplementary materials submitted with the paper.

4.1 Syntax

Figure 1 shows the syntax of $\mathsf{F_i}$. The differences to λ_i are highlighted in gray.

---

[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 $\mathsf{F_i}$

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

Fig. 1. F_i syntax.

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.

4.2 Subtyping

The subtyping rules of the form $A <: B$ are shown in Figure 2. 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. $S\alpha$ defines subtyping as a reflexive relation on

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

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

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

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

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

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

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

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

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

Fig. 2. Subtyping rules of F_i.

type variables. In S$\forall$ a universal quantifier ($\forall$) is covariant in its body, and contravariant in its disjointness constraints. The SR{\small EC} rule says that records 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 2, 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 1 (Subtyping reflexivity). *For any type* A, A <: A.

Proof. By induction on A.

BRUNO: too much white space between Lemma and proof

Lemma 2 (Subtyping transitivity). *If* $A <: B$ *and* $B <: C$, *then* $A <: C$.

Proof. By double induction on both derivations.

BRUNO: Too much space waisted between Lemma and proof. reduce the white space.

4.3 Typing

Well-formedness The well-formedness rules are shown in the top part of Figure 3. 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 3. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Wellformedness and type system of $\mathsf{F_i}$.

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

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

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

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

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

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

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

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

Fig. 4. Algorithmic Disjointness.

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 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 $\mathsf{F_i}$, 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 4, 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. $\mathsf{D}\top$ and $\mathsf{D}\top\mathsf{Sym}$ say that any type is disjoint to $\top$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: $\mathsf{D}\alpha$ is the base rule; and $\mathsf{D}\alpha\mathsf{Sym}$ is its twin symetrical 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 3 (Covariance of disjointness).
   If $\Gamma \vdash \mathsf{A} * \mathsf{B}$ *and* $\mathsf{B} <: \mathsf{C}$, *then* $\Gamma \vdash \mathsf{A} * \mathsf{C}$.

   BRUNO: Did you prove it?
   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 $\mathsf{x} * \mathsf{x}$, 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 $\mathsf{D}\forall$ 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:

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

The question is under which conditions are those two types disjoint. In the first type α cannot be instantiated with $\mathtt{Int}$ and in the second case α cannot be instantiated with $\mathtt{Char}$. Therefore for both bodies to be disjoint, α cannot be instantiated with either $\mathtt{Int}$ or $\mathtt{Char}$. 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 $\mathsf{D}\to$ says that two function types are disjoint if and only if their return types are disjoint. The rules dealing with intersection types ($\mathsf{D}\&\mathsf{L}$ and $\mathsf{D}\&\mathsf{R}$) 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 $\mathsf{A} *_{\mathrm{ax}} \mathsf{B}$ 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 $\mathsf{D}\alpha$ and $\mathsf{D}\alpha\mathsf{Sym}$, and disjointness with the $\top$ type is already captured by $\mathsf{D}\top$ and $\mathsf{D}\top\mathsf{Sym}$.

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 $\forall(A * \mathtt{Int}).(\mathtt{Int}\&A) \to \mathtt{Int}$, 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 $\alpha * \mathtt{Int}$, α and $\mathtt{Int}$ are disjoint:

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

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

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

is no longer true since $\mathtt{Int}$ 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 4 (Disjointness is stable under substitution).
$\quad$ *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 5 (Types are stable under substitution).
    If $\Gamma \vdash A$ and $\Gamma \vdash B$ and $(x * C) \in \Gamma$ and $\Gamma \vdash B * C$, then $[x := B]\ \Gamma \vdash [x := B]\ A$.

BRUNO: In several places in this section you are using x as a type variable, when you should instead be using α. You need to be consistent with your own notation: you defined x to be a (term) variable and α to be a type-variable.

Proof. By induction on the well-formedness derivation of A. The intersection case requires the use of Lemma 4. Also, the variable case required proving that if x 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 6 (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 5 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 3, which was used to justify the rule for disjointness of variables. If one takes A as some variable x, then the lemma should read as:

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

Now we can ask: how many suitable types are there to instantiate x 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 x, since x must be disjoint to all possible C's. Thus, to answer the first question, the options for instantiating x 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 x with the most options for instantiation whereas $\bot$ will deliver the least options. We will briefly discuss the role of the former in our system. JOAO: maybe we can just refer to the overview and delete the next paragraph? BRUNO: That could be an idea, yes.

The most liberal bound. It is easy to see that $\top$ is the most liberal type since it is disjoint to everything. This means that the $\top$ type actually plays an important role in our system, since the latter must be complete with respect to System F. In other words, any program accepted by System F should also be accepted by $\mathsf{F_i}$. Since System F does not contain disjointness quantification, $\top$ comes in handy: the System F's type $\forall \alpha. \mathsf{T_F}$ (where $\mathsf{T_F}$ is some other type), is equivalent to $\mathsf{F_i}$'s type $\forall (\alpha * \top). \mathsf{T_i}$, where $\mathsf{T_i}$ is also an equivalent translation of $\mathsf{T_F}$. An example of such translation has been given in Section 2.

6   Semantics and Coherence

This section discusses the elaboration semantics of $\mathsf{F_i}$ and show how coherence is retained. It will first explain the semantics by means of the elaboration to System F. Then, it will discuss the necessary extensions to retain coherence, namely in the coercions of top-like types; coercive subtyping, and bidirectional type-system's elaboration.

6.1   Semantics

The dynamic semantics of the call-by-value $\mathsf{F_i}$ are defined by means of a type-directed translation to an extension of System F with pairs [6].

Target language. The syntax and typing of our target language is unsurprising. The syntax of the target language is shown in Figure 5. The highlighted part shows its difference with the standard System F. The typing rules can be found in appendix. JOAO: actually they can't, but should we include them?

Type and context translation. Figure 6 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.

---

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

[6] For simplicity, we will just refer to this system as "System F" from now on.

$$\text{Types} \quad T ::= \alpha \mid \texttt{Int} \mid T_1 \to T_2 \mid \forall\alpha.\,T$$
$$\mid\ () \mid\ (T_1, T_2)$$
$$\text{Terms} \quad E ::= x \mid i \mid \lambda x.\,E \mid E_1\ E_2 \mid \Lambda\alpha.\,E \mid E\ T$$
$$\mid\ () \mid\ (E_1, E_2) \mid \texttt{proj}_k E \quad k \in \{1, 2\}$$
$$\text{Contexts}\ G ::= \cdot \mid G, \alpha \mid G, x{:}T$$

Fig. 5. Target language syntax.

$$\boxed{|A| = T}$$

$$|\alpha| = \alpha$$
$$|\top| = ()$$
$$|A_1 \to A_2| = |A_1| \to |A_2|$$
$$|\forall(\alpha * A).\,B| = \forall\alpha.\,|B|$$
$$|A_1 \& A_2| = (|A_1|, |A_2|)$$

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

$$|\cdot| = \cdot$$
$$|\Gamma, \alpha * A| = |\Gamma|, \alpha$$
$$|\Gamma, \alpha{:}A| = |\Gamma|, \alpha{:}|A|$$

Fig. 6. Type and context translation.

6.2 Top-like types and their coercions

Our definition of top-like types is naturally extended from λ_i. The rules that compose this unary relation, denoted as $\rceil.\lceil$, are presented at the top of Figure 7. The only new rule is TOPLIKE-FORALL, which extends the notion of top-like types for the (disjoint) quantifier case.

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 7. 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.

$$\boxed{\rceil A \lceil}$$

$$\frac{}{\rceil \top \lceil} \ \text{TOPLIKE-TOP} \qquad \frac{\rceil A \lceil \quad \rceil B \lceil}{\rceil A\&B \lceil} \ \text{TOPLIKE-INTER} \qquad \frac{\rceil B \lceil}{\rceil A \to B \lceil} \ \text{TOPLIKE-FUN}$$

$$\frac{\rceil A \lceil}{\rceil \forall (\alpha * B).\, A \lceil} \ \text{TOPLIKE-FORALL}$$

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

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

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

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

Fig. 7. Top-like types and their coercions.

6.3 Coercive Subtyping and Coherence

Coercive subtyping. The judgement

$$A_1 <: A_2 \ \boxed{\hookrightarrow E}$$

extends the subtyping judgement in Figure 2 with a coercion on the right hand side of $\boxed{\hookrightarrow}$. A coercion E is just an term in the target language and is ensured to have type $|A_1| \to |A_2|$ (by Lemma 7). For example,

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

generates a coercion function with type: $\texttt{Int}\&\alpha \to \alpha$.

Rule $S\top$ the coercion is the constant function of the unit term. In rules $S\alpha$, $S\mathbb{Z}$, 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 7 (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, we should prove that our subtyping relation is also coherent. More formally, with disjoint polymorphism the following theorem holds:

Lemma 8 (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.

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

Lemma 9 (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-system. 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 [18]. It also translates the form of disjoint quantification and disjoint type application into regular (polymorphic) quantification and type application. For example,

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

in F_i will be translated into System F's:

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

This is achieved by means of the type-directed translation. Note that, when performing type application, the type-system will ensure that the type being applied is compatible with the disjoint constraint, otherwise the program is rejected. For example, applying the term above to `Char` would be accepted, and the generated term would translate this into System F's type application. On the other hand, applying the term to `Int` would be rejected, and hence the target term produced is meaningless.

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 will look into the coercions of the new rules in greater detail. Rule T-BLAM introduces a type variable a into context, and naturally does not make use of the disjointness constraint. In F_i, one may pass more information to a function than what is required; but not in System F. To account for this difference, in T-APP, the coercion E from the subtyping relation is applied to the argument. Rules T-REC and T-PROJR are quite straightforward, since there are not records in the target language. 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.)

BRUNO: Yes, and you want to mention somewhere where the stability of substitution lemma plays a role. I presume it plays a role in the coherence theorems.
JOAO: from what I could gather we use it to ensure well-formed typing, which is used here (as mentioned below)

Proof. By induction on the first derivation. Note that three cases need special attention: T-TAPP, T-SUB and T-APP. In the T-BLAM:

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

we need to show that the constraint B is unique, using Theorem 3, in order to apply the induction hypothesis. For T-APP:

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

we also need to show the uniqueness of A_1 using Theorem 3, in order to apply the induction hypothesis. Also, in the T-SUB rule:

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

we need to show that E_{sub} is unique (by Lemma 9), and thus to show that A is well-typed (by Lemma 6). 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

Coherence Reynolds invented Forsythe [31] in the 1980s. Our merge operator is analogous to his operator p_1, p_2. Forsythe has a coherent semantics. The result was proved formally by Reynolds [30] in a lambda calculus with intersection types and a merge operator. However there are two key differences to our work. Firstly the way coherence is ensured is rather ad-hoc. He has four different typing rules for the merge operator, each accounting for various possibilities of what the types of the first and second components are. In some cases the meaning of the second

component takes precedence (that is, is biased) over the first component. The set of rules is restrictive and it forbids, for instance, the merge of two functions (even when they a provably disjoint). In contrast, disjointness in F_i^* has a well-defined specification and it is quite flexible. Secondly, Reynolds calculus does not support universal quantification. It is unclear to us whether his set of rules would still ensure disjointness in the presence of universal quantification. Since some biased choice is allowed in Reynold's calculus the issues illustrated in Section **??** could be a problem.

Pierce [26] made a comprehensive review of coherence, especially on Curien and Ghelli [13] and Reynolds' methods of proving coherence; but he was not able to prove coherence for his $\mathsf{F}_\wedge$ calculus. He introduced a primitive `glue` function as a language extension which corresponds to our merge operator. However, in his system users can "glue" two arbitrary values, which can lead to incoherence.

Our work is largely inspired by Dunfield [18]. He described a similar approach to ours: compiling a system with intersection types and a merge operator into ordinary λ-calculus terms with pairs. One major difference is that his system does not include parametric polymorphism, while ours does not include unions. The calculus presented in Section 4 can be seen as a relatively straightforward extension of Dunfield's calculus with parametric polymorphism. However, as acknowledged by Dunfield, his calculus lacks of coherence. He discusses the issue of coherence throughout his paper, mentioning biased choice as an option (albeit a rather unsatisfying one). He also mentioned that the notion of disjoint intersection could be a good way to address the problem, but he did not pursue this option in his work. In contrast to his work, we developed a type system with disjoint intersection types and proposed disjoint quantification to guarantee coherence in our calculus.

Intersection types with polymorphism. Our type system combines intersection types and parametric polymorphism. Closest to us is Pierce's work [27] on a prototype compiler for a language with both intersection types, union types, and parametric polymorphism. Similarly to F_i^* in his system universal quantifiers do not support bounded quantification. However Pierce did not try to prove any meta-theoretical results and his calculus does not have a merge operator. Pierce also studied a system where both intersection types and bounded polymorphism are present in his Ph.D. dissertation [26] and a 1997 report [28].

Going in the direction of higher kinds, Compagnoni and Pierce [10] added intersection types to System F_ω and used the new calculus, $\mathsf{F}_\wedge^\omega$, to model multiple inheritance. In their system, types include the construct of intersection of types of the same kind K. Davies and Pfenning [15] studied the interactions between intersection types and effects in call-by-value languages. And they proposed a "value restriction" for intersection types, similar to value restriction on parametric polymorphism. Although they proposed a system with parametric polymorphism, their subtyping rules are significantly different from ours, since they consider parametric polymorphism as the "infinit analog" of intersection polymorphism.

Recently, Castagna et al. [9] studied an very expressive calculus that has polymorphism and set-theoretic type connectives (such as intersections, unions, and negations). As a result, in their calculus one is also able to express a type variable that can be instantiated to any type other than $\mathtt{Int}$ as $\alpha \setminus \mathtt{Int}$, which is syntactic sugar for $\alpha \wedge \neg\mathtt{Int}$. As a comparison, such a type will need a disjoint quantifier, like $\forall(\alpha * \mathtt{Int}).\alpha$, in our system. Unfortunately their calculus does not include a merge operator like ours.

There have been attempts to provide a foundational calculus for Scala that incorporates intersection types [2,1]. Although the minimal Scala-like calculus does not natively support parametric polymorphism, it is possible to encode parametric polymorphism with abstract type members. Thus it can be argued that this calculus also supports intersection types and parametric polymorphism. However, the type-soundness of a minimal Scala-like calculus with intersection types and parametric polymorphism is not yet proven. Recently, some form of intersection types has been adopted in object-oriented languages such as Scala, Ceylon, and Grace. Generally speaking, the most significant difference to F_i^* is that in all previous systems there is no explicit introduction construct like our merge operator. As shown in Section **??**, this feature is pivotal in supporting modularity and extensibility because it allows dynamic composition of values.

Other type systems with intersection types. Refinement intersection [17,14,19] is the more conservative approach of adopting intersection types. It increases only the expressiveness of types but not terms. But without a term-level construct like "merge", it is not possible to encode various language features. As an alternative to syntactic subtyping described in this paper, Frisch et al. [20] studied semantic subtyping. Semantic subtyping seems to have important advantages over syntactic subtyping. One worthy avenue for future work is to study languages with intersection types and merge operator in a semantic subtyping setting.

Extensibility. One of our motivations to study systems with intersections types is to better understand the type system requirements needed to address extensibility problems. A well-known problem in programming languages is the Expression Problem [38]. In recent years there have been various solutions to the Expression Problem in the literature. Mostly the solutions are presented in a specific language, using the language constructs of that language. For example, in Haskell, type classes [39] can be used to implement type-theoretic encodings of datatypes [21]. It has been shown [7] that, when encodings of datatypes are modeled with type classes, the subclassing mechanism of type classes can be used to achieve extensibility and reuse of operations. Using such techniques provides a solution to the Expression Problem. Similarly, in OO languages with generics, it is possible to use generic interfaces and classes to implement type-theoretic encodings of datatypes. Conventional subtyping allows the interfaces and classes to be extended, which can also be used to provide extensibility and reuse of operations. Using such techniques, it is also possible to solve the Expression Problem in OO languages [32,24]. It is even possible to solve the Expression Problem in theorem provers like Coq, by exploiting Coq's type class mechanism [16].

Nevertheless, although there is a clear connection between all those techniques and type-theoretic encodings of datatypes, as far as we know, no one has studied the expression problem from a more type-theoretic point of view. As shown in Section **??**, a system with intersection types, parametric polymorphism, the merge operator and disjoint quantification can be used to explain type-theoretic encodings with subtyping and extensibility.

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

Encoding records using intersection types appeared in Reynolds [31] and Castagna et al. [8]. Although Dunfield also discussed this idea in his paper [18], he only provided an implementation but not a formalization. Very similar to our treatment of elaborating records is Cardelli's work [4] on translating a calculus, named $F_{<:\rho}$, with extensible records to a simpler calculus that without records primitives (in which case is $F_{<:}$). 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 [40] started the work on extensible records and proposed row types [41] for records. Cardelli and Mitchell [6] defined three primitive operations on records that are similar to ours: *selection*, *restriction*, and *extension*. The merge operator in F_i plays the same role as extension. Following Cardelli and Mitchell's approach, of restriction and extension. Both Leijen's systems [22,23] and ours allow records that contain duplicate labels. Leijen's system is more sophisticated. For example, it supports passing record labels as arguments to functions. He also showed an encoding of intersection types using first-class labels.

8   Conclusion and Future Work

This paper described F_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. Amin, N., Moors, A., Odersky, M.: Dependent object types. In: 19th International Workshop on Foundations of Object-Oriented Languages (2012)
2. Amin, N., Rompf, T., Odersky, M.: Foundations of path-dependent types. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (2014)
3. Boehm, C., Berarducci, A.: Automatic synthesis of typed lambda-programs on term algebras. Theoretical Computer Science 39, 135–154 (1985)
4. Cardelli, L.: Extensible records in a pure calculus of subtyping. Digital. Systems Research Center (1992)
5. Cardelli, L., Martini, S., Mitchell, J.C., Scedrov, A.: An extension of system f with subtyping. Inf. Comput. 109(1-2) (Feb 1994)
6. Cardelli, L., Mitchell, J.C.: Operations on records. In: Mathematical foundations of programming semantics (1990)
7. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. 19(5) (2009)
8. Castagna, G., Ghelli, G., Longo, G.: A calculus for overloaded functions with subtyping. Information and Computation (1995)
9. Castagna, G., Nguyen, K., Xu, Z., Im, H., Lenglet, S., Padovani, L.: Polymorphic functions with set-theoretic types: Part 1: Syntax, semantics, and evaluation. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '14 (2014)
10. Compagnoni, A.B., Pierce, B.C.: Higher-order intersection types and multiple inheritance. Mathematical Structures in Computer Science (1996)
11. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional characters of solvable terms. Mathematical Logic Quarterly (1981)
12. Crary, K.: Simple, efficient object encoding using intersection types. Tech. rep., Cornell University (1998)
13. Curienl, P.L., Ghelli, G.: Coherence of subsumption. In: CAAP'90: 15th Colloquium on Trees in Algebra and Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings. vol. 431, p. 132. Springer Science & Business Media (1990)
14. Davies, R.: Practical refinement-type checking. Ph.D. thesis, University of Western Australia (2005)
15. Davies, R., Pfenning, F.: Intersection types and computational effects. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00) (2000)
16. Delaware, B., d. S. Oliveira, B.C., Schrijvers, T.: Meta-theory à la carte. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25 (2013)
17. Dunfield, J.: Refined typechecking with stardust. In: Proceedings of the 2007 workshop on Programming languages meets program verification. ACM (2007)
18. Dunfield, J.: Elaborating intersection and union types. Journal of Functional Programming (2014)
19. Freeman, T., Pfenning, F.: Refinement types for ml. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. PLDI '91 (1991)
20. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM (JACM) (2008)

21. Hinze, R.: Generics for the masses. J. Funct. Program. 16(4-5) (Jul 2006)
22. Leijen, D.: First-class labels for extensible rows. UU-CS (2004-051) (2004)
23. Leijen, D.: Extensible records with scoped labels. Trends in Functional Programming (2005)
24. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses. In: ECOOP 2012–Object-Oriented Programming (2012)
25. 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)
26. Pierce, B.C.: Programming with intersection types and bounded polymorphism. Ph.D. thesis, Carnegie Mellon University Pittsburgh, PA (1991)
27. Pierce, B.C.: Programming with intersection types, union types, and polymorphism (1991)
28. Pierce, B.C.: Intersection types and bounded polymorphism. Mathematical Structures in Computer Science (1997)
29. Rendel, T., Brachthäuser, J.I., Ostermann, K.: From object algebras to attribute grammars. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '14 (2014)
30. 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)
31. Reynolds, J.C.: Design of the programming language Forsythe (1997)
32. d. S. Oliveira, B.C.: Modular visitor components: A practical solution to the expression families problem. In: 23rd European Conference on Object Oriented Programming (ECOOP) (2009)
33. Bruno C. d. S. Oliveira, R.H., Loeh, A.: Extensible and modular generics for the masses. In: Nilsson, H. (ed.) Trends in Functional Programming (2006)
34. Schwaab, C., Siek, J.G.: Modular type-safety proofs in agda. In: Proceedings of the 7th Workshop on Programming languages meets program verification (PLPV) (2013)
35. Swierstra, W.: Data types à la carte. J. Funct. Program. 18(4) (Jul 2008)
36. Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423–436 (July 2008)
37. Torgersen, M.: The Expression Problem Revisited. In: Odersky, M. (ed.) Proc. of the 18th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science (2004)
38. Wadler, P.: The expression problem. Java-genericity mailing list (1998)
39. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: POPL. pp. 60–76. ACM (1989)
40. Wand, M.: Complete type inference for simple objects. In: LICS (1987)
41. 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)
42. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: FOOL (2005)