Disjoint Polymorphism

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

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

Abstract. Dunfield has shown that a simply typed core calculus with intersection types and a merge operator forms a powerful foundation for various programming language features. While his calculus is type-safe, it lacks *coherence*: different derivations for the same expression can lead to different results. The lack of coherence is important disadvantage for adoption of his core calculus in implementations of programming languages, as the semantics of the programming language becomes implementation dependent. Moreover his calculus did not account for parametric polymorphism.

This paper presents $\mathsf{F}_{\mathsf{i}}^{*}$: a core calculus with a variant of *intersection types*, *parametric polymorphism* and a *merge operator*. The semantics $\mathsf{F}_{\mathsf{i}}^{*}$ is both type-safe and coherent. Coherence is achieved by ensuring that intersection types are *disjoint*. Formally two types are disjoint if they do not share a common supertype. We present a type system that prevents intersection types that are not disjoint, as well as an algorithmic specification to determine whether two types are disjoint. Moreover we show that this approach extends to systems with parametric polymorphism. Parametric polymorphism makes the problem of coherence significantly harder. When a type variable occurs in an intersection type, it is not statically known whether the instantiated type will share a common supertype with other components of the intersection. To address this problem we propose *disjoint quantification*: a constrained form of parametric polymorphism, that allows programmers to specify disjointness constraints for type variables. With disjoint quantification the calculus remains very flexible in terms of programs that can be written with intersection types, while retaining coherence.

1 Introduction

Previous work by Dunfield [18] has shown the power of intersection types and a merge operator. The presence of a merge operator in a core calculus provides significant expressiveness, allowing encodings for many other language constructs as syntactic sugar. For example single-field records are easily encoded as types with a label, and multi-field records are encoded as the concatenation of single-field records. Concatenation of records is expressed using intersection types at the type-level and the corresponding merge operator at the term level. Dunfield formalized a simply typed lambda calculus with intersection types and a merge operator. He showed how to give a semantics to the calculus by a type-directed

translation to a simply typed lambda calculus extended with pairs. The type-directed translation is simple, elegant, and type-safe.

Intersection types and the merge operator are also useful in the context of software *extensibility*. In recent years there has been a wide interest in presenting solutions to the *expression problem* [36] in various communities. Currently there are various solutions in functional programming languages [34,32], object-oriented programming languages [35,40,31,24] and theorem provers [16,33]. Many of the proposed solutions for extensibility are closely related to type-theoretic encodings of datatypes [3], except that some form of subtyping is also involved. Various language-specific mechanisms are used to combine ideas from type-theoretic encodings of datatypes with subtyping, but the essence of the solutions is hidden behind the peculiarities of particular programming languages. Calculi with intersection types have a natural subtyping relation that is helpful to model problems related to extensibility. Moreover, intersection types and an *encoding* of a merge operator have been shown to be useful to solve additional challenges related to extensibility [25]. Therefore it is natural to wonder if a core calculus supporting parametric polymorphism, intersection types and a merge operator, can be used to capture the essence of various solutions to extensibility problems.

Dunfield calculus seems to provide a good basis for a foundational calculus for studying extensibility. However, his calculus is still insufficient for studying extensibility for two different reasons. Firstly it does not support parametric polymorphism. This is a pressing limitation because type-theoretic encodings of datatypes fundamentally rely on parametric polymorphism. Secondly, and more importantly, while Dunfield calculus is type-safe, it lacks the property of *coherence*: different derivations for the same expression can lead to different results. The lack of coherence is an important disadvantage for adoption of his core calculus in implementations of programming languages, as the semantics of the programming language becomes implementation dependent. Moreover, from the theoretic point-of-view, the ambiguity that arises from the lack of coherence makes the calculus unsatisfying when the goal is to precisely capture the essence of solutions to extensibility.

This paper presents $\mathsf{F}_{\mathsf{i}}^{*}$: a core calculus with a variant of *intersection types*, *parametric polymorphism* and a *merge operator*. The semantics $\mathsf{F}_{\mathsf{i}}^{*}$ is both type-safe and coherent. Thus $\mathsf{F}_{\mathsf{i}}^{*}$ addresses the two limitations of Dunfield calculus and can be used to express the key ideas of extensible type-theoretic encodings of datatypes.

Coherence is achieved by ensuring that intersection types are *disjoint*. Given two types A and B, two types are disjoint ($\mathsf{A} * \mathsf{B}$) if there is no type C such that both A and B are subtypes of C. Formally this definition is captured as follows:

$$\mathsf{A} * \mathsf{B} \equiv \nexists \mathsf{C}.\ \mathsf{A} <: \mathsf{C} \wedge \mathsf{B} <: \mathsf{C}$$

With this definition of disjointness we present a formal specification of a type system that prevents intersection types that are not disjoint. However, the formal definition of disjointness does not lend itself directly to an algorithmic implementation. Therefore, we also present an algorithmic specification to determine

whether two types are disjoint. Moreover, this algorithmic specification is shown to be sound and complete with respect to the formal definition of disjointness.

Disjoint intersection types can be extended to support parametric polymorphism. However, parametric polymorphism makes the problem of coherence significantly harder. When a type variable occurs in an intersection type, it is not statically known whether the instantiated type will share a common supertype with other components of the intersection. To address this problem we propose *disjoint quantification*: a constrained form of parametric polymorphism, that allows programmers to specify disjointness constraints for type variables. With disjoint quantification the calculus remains very flexible in terms of programs that can be written with intersection types, while retaining coherence.

We also investigate how to do type-theoretic encodings of datatypes in F_{i}. In particular it is shown that extensions of datatype encodings have subtyping relations with the datatype they extend. Moreover, it is possible to reuse code from the operations on the original datatype and consequently solve the Expression Problem. Finally, it is shown how *all the features* of F_{i} (intersection types, the merge operator, parametric polymorphism and disjoint quantification) are needed to properly encode one important combinator [25] used to compose multiple operations over datatypes.

In summary, the contributions of this paper are:

- **Disjoint Intersection Types:** A new form of intersection type where only disjoint types are allowed. A sound and complete algorithmic specification of disjointness (with respect to the corresponding formal definition) is presented.
- **Disjoint Quantification:** A novel form of universal quantification where type variables can have disjointness constraints.
- **Formalization of System $\mathsf{F}_{\mathsf{i}}^{*}$ and Proof of Coherence:** An elaboration semantics of System $\mathsf{F}_{\mathsf{i}}^{*}$ into System F is given. Type-soundness and coherence are proved.
- **Extensible Type-Theoretic Encodings:** We show that in $\mathsf{F}_{\mathsf{i}}^{*}$ type-theoretic encodings can be combined with subtyping to provide extensibility.
- **Implementation:** An implementation of an extension of System $\mathsf{F}_{\mathsf{i}}^{*}$, as well as the examples presented in the paper, are publicly available[1].

2 Overview

This section introduces $\mathsf{F}_{\mathsf{i}}^{*}$ 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 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 $\mathsf{F}_{\mathsf{i}}^{*}$. Although

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

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)
```

If a value `x` has type `Int & Char` then `x` can be used as an integer or as a character. Therefore, `x` can be used as an argument to any function that takes an integer as an argument, or any function that take a character as an argument. In the program above the functions `idInt` and `idChar` are the identity functions on integers and characters, respectively. Passing `x` as an argument to either one (or both) of the functions is valid.

Merge operator. In the previous program we deliberately did not show how to introduce 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*. 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 operator and pairs. The merge operator is similar to the introduction construct on pairs. An analogous implementation of `x` with pairs would be:

```
let xPair : (Int, Char) = (1, 'c') in ...
```

The significant difference between intersection types with a merge operator and pairs is in the elimination construct. With pairs there are explicit eliminators (`fst` and `snd`). These eliminators must be used to extract the components of the right type. For example, in order to use `idInt` and `idChar` with pairs, we would need to write a program such as:

```
(idInt (fst xPair), idChar (snd xPair))
```

In contrast the elimination of intersection types is done implicitly, by following a type-directed process. For example, when a value of type `Int` is needed, but an intersection of type `Int & Char` is found, the compiler uses the type system to extract the corresponding value.

2.2 Incoherence

Unfortunatelly the implicit nature of elimination for intersection types built with a merge operator can lead to incoherence. The merge operator combines two terms, of type A and B respectively, to form a term of type A&B. For example, $1,,`c'$ is of type `Int`&`Char`. In this case, no matter if $1,,`c'$ is used as `Int` or `Char`, the result of evaluation is always clear. However, with overlapping types, it is not straightforward anymore to see the result. For example, what should be the result of this program, which asks for an integer out of a merge of two integers:

```
(fun (x: Int) → x) (1,,2)
```

Should the result be `1` or `2`?

If both results are accepted, we say that the semantics is *incoherent*: there are multiple possible meanings for the same valid program. Dunfield's calculus [18] is incoherent and accepts the program above.

Getting around incoherence: biased choice. In a real implementation of Dunfield calculus a choice has to be made on which value to compute. For example, one potential option is to always take the left-most value matching the type in the merge. Similarly, one could always take the right-most value matching the type in the merge. Either way, the meaning of a program will depend on a biased implementation choice, which is clearly unsatisfying from the theoretical point of view (although perhaps acceptable in practice). Moreover, even if we accept a particular biased choice as being good enough, the approach cannot be easily extended to systems with parametric polymorphism, as we illustrate in Section 2.4.

2.3 Restoring Coherence: Disjoint Intersection Types

Coherence is a desirable property for a semantics. A semantics is said to be coherent if any *valid program* has exactly one meaning [29] (that is, the semantics is not ambiguous). One option to restore coherence is to reject programs which may have multiple meanings. Analyzing the expression $1,,2$, we can see that the reason for incoherence is that there are multiple, overlapping, integers in the merge. Generally speaking, if both terms can be assigned some type C, both of them can be chosen as the meaning of the merge, which leads to multiple meanings of a term. Thus a natural option is to try to forbid such overlapping values of the same type in a merge.

This is precisely the approach taken in F^*_i. F^*_i requires that the two types of in intersection must be *disjoint*. However, although disjointness seems a natural

restriction to impose on intersection types, it is not obvious to formalize it. Indeed Dunfield has mentioned disjointness as an option to restore coherence, but he left it for future work due to the non-triviality of the approach.

Searching for a definition of disjointness. The first step towards disjoint intersection types is to come up with a definition of disjointness. A first attempt at such definition would be to require that, given two types A and B, both types are not subtypes of each other. Thus, denoting disjointness as $\mathsf{A} * \mathsf{B}$, we would have:

$$\mathsf{A} * \mathsf{B} \equiv \mathsf{A} \not<: \mathsf{B} \wedge \mathsf{B} \not<: \mathsf{A}$$

At first sight this seems a reasonable definition and it does prevent merges such as `1,,2`. However some moments of thought are enough to realize that such definition does not ensure disjointness. For example, consider the following merge:

```
((1,,'c') ,, (2,,True))
```

This merge has two components which are also intersection types. The first component (`(1,,'c')`) has type `Int&Char`, whereas the second component (`(2 ,, True)`) has type `Int&Bool`. Clearly,

$$\texttt{Int\&Char} \not<: \texttt{Int\&Bool} \wedge \texttt{Int\&Bool} \not<: \texttt{Int\&Char}$$

Nevertheless the following program still leads to incoherence:

```
(fun (x: Int) → x) ((1,,'c'),,(2,,True))
```

as both `1` or `2` are possible outcomes of the program. Although this attempt to define disjointness failed, it did bring us some additional insight: although the types of the two components of the merge are not subtypes of each other, they share some types in common.

A proper definition of disjointness. In order for two types to be trully disjoint, they must not have any subcomponents sharing the same type. In a system with intersection types this can be ensured by requiring the two types do not share a common supertype. The following definition captures this idea more formally.

Definition 1 (Disjointness). *Given two types* A *and* B*, two types are disjoint (written* $\mathsf{A} * \mathsf{B}$*) if there is no type* C *such that both* A *and* B *are subtypes of* C*:*

$$\mathsf{A} * \mathsf{B} \equiv \nexists \mathsf{C}.\ \mathsf{A} <: \mathsf{C} \wedge \mathsf{B} <: \mathsf{C}$$

This definition of disjointness prevents the problematic merge. Since `Int` is a common supertype of both `Int&Char` and `Int&Bool`, those two types are not disjoint.

$\mathsf{F}^{*}_{\mathsf{i}}$'s type system only accepts programs that use disjoint intersection types. As shown in Section 6 disjoint intersection types will play a crutial rule in guaranteeing that the semantics is coherent.

2.4 Parametric Polymorphism and Intersection Types

Before we show how F_i^* extends the idea of disjointness to parametric polymorphism, we discuss some non-trivial issues that arise from the interaction between parametric polymorphism and intersection types. Consider the attempt to write the following polymorphic function in F_i^* (we use uppercase Latin letters to denote type variables):

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

The `fst` function is supposed to extract a value of type (`A`) from the merge value `x` (of type `A&B`). However this function is problematic. The reason is that 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 issue incoherence would be to bias the type-based merge lookup to the left or to the right (as discussed in Section 2.2). Unfortunately, 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: Int) → z) (1,,2) --  By the definition of fst
⇝ (fun (z: Int) → z) 2      --  Right-biased coercion
⇝ 2                         --  By β-reduction
```

However (assumming 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. Clearly this is inconsistent with the biased choice!

Unfortunately this 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^*.

Conservatively rejecting intersections. To avoid incoherence, and the issues of biased choice, another option is simply to reject programs where the instantiations of type variables may lead to incoherent programs. In this case the definition of `fst` would be rejected, since there are indeed some cases that may lead to incoherent programs. Unfortunately this is too restrictive and prevents many useful programs using both parametric polymorphism and intersection types. In particular, in the case of `fst`, if the two type parameters are used with two disjoint intersection types, then the merge will not lead to ambiguity.

In summary, it seems hard to have parametric polymorphism, intersection types and coherence without being overly conservative.

2.5 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) = (fun (z: A) → z) x
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.

The nice thing about this solution is that many uses of `fst` are 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`. However, problematic uses of `fst` are rejected. For example:

```
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`.

3 Application: Extensible records

JOAO: change syntax to the one used in overview Our system can be used to encode records, similarly to way as discussed in [18]. However, describing and

implementing records within programming languages is certainly not novel and has been extensively studied in the past. Most of the systems are entirely focused on concrete aspects of records (i.e. expressiveness, compilation, etc), while ours will specialize 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 these basic operations in the context of F_i.

3.1 Basic operations

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

$$\begin{aligned} &\text{select} :: \forall(\alpha * \top).\{\mathsf{l} : \alpha\} \to \alpha \\ &\text{select} \ = \Lambda(\alpha * \top).\lambda x.\, x.\mathsf{l} \end{aligned}$$

Note how, through the use of subtyping, this function will accept any intersection type that contains the single recorld $\{\mathsf{l} : \alpha\}$. This resembles other systems ..., 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:

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

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, 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):

$$\begin{aligned} &\text{addL}_1 :: \forall\alpha \# \mathsf{l}.\alpha \to (\alpha \| \{\mathsf{l} : \texttt{Int}\}) \\ &\text{addL}_1 \ = \ldots \end{aligned}$$

The reader might notice the resemblance with our system:

$$\begin{aligned}&\mathrm{addL}_2 :: \forall(\alpha * \{\mathtt{l} : \mathtt{Int}\}).\, \alpha \to (\alpha \& \{\mathtt{l} : \mathtt{Int}\})\\&\mathrm{addL}_2\ = \ldots\end{aligned}$$

This shows that one can use disjoint quantification to express negative field information. It is very close to what (Harper & Pierce) describe in their system. Note how we have 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 $\mathtt{l}$, with a field type *disjoint* to $\mathtt{Int}$.

Had one meant to forbid records with *any* $\mathtt{l}$ fields, then one could write:
JOAO: how about this? fresh beta vs bottom?

$$\begin{aligned}&\mathrm{addL}_3 :: \forall(\beta * \top).\, \forall(\alpha * \{\mathtt{l} : \beta\}).\, \alpha \to (\alpha \& \{\mathtt{l} : \beta\})\\&\mathrm{addL}_3\ = \ldots\end{aligned}$$

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.

3.2 Expressibility

In ... (SPJ & MJ) – a strict system with extension – an example of a function that uses record types is the following:

$$\begin{aligned}&\mathrm{average}_1 :: (\mathtt{r}\backslash\mathtt{y}, \mathtt{r}\backslash\mathtt{x}) \Rightarrow \{\mathtt{r}|\mathtt{x} :: \mathtt{Int}, \mathtt{y} :: \mathtt{Int}\} \to \mathtt{Int}\\&\mathrm{average}_1\ \mathtt{r} = (\mathtt{r.x} + \mathtt{r.y})/2\end{aligned}$$

The type signature says that for any record with type $\mathtt{r}$, that lacks both $\mathtt{x}$ $\mathtt{y}$, can be accepted as parameter extended with $\mathtt{x}$ $\mathtt{y}$, returning an integer. Note how the bounded polymorphism is essential to ensure that $\mathtt{r}$ does not contain $\mathtt{x}$ nor $\mathtt{y}$. On the other hand, in a system with free extension as in [23], the more general program would be accepted:

$$\begin{aligned}&\mathrm{average}_2 :: \forall \mathtt{x}\ \mathtt{y}, \{\mathtt{x} :: \mathtt{Int}, \mathtt{y} :: \mathtt{Int}|\mathtt{r}\} \to \mathtt{Int}\\&\mathrm{average}_2\ \mathtt{r} = (\mathtt{r.x} + \mathtt{r.y})/2\end{aligned}$$

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

In $\mathsf{F}_\mathtt{i}$, such function could be re-written as [2]:

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

Thus more types are accepted this function than in the first system, but less than the second. Another major difference between $\mathsf{F}_\mathtt{i}$ 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?) $\mathsf{F}_\mathtt{i}$ 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]:

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

4 The $\mathsf{F}_\mathtt{i}$ Calculus

This section presents the syntax, subtyping, and typing of $\mathsf{F}_\mathtt{i}$: a calculus with intersection types, parametric polymorphism, records and a merge operator. This calculus is an extension of $\lambda_\mathtt{i}$ and Dunfield's calculus [18], which are simply typed calculus with intersection types and a merge operator. Section 6 introduces the necessary changes to the definition of disjointness in order to retain coherence.

All the meta-theory 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 $\mathsf{F}_\mathtt{i}$. The differences to $\lambda_\mathtt{i}$ are highlighted in gray.

Types. Metavariables A, B range over types. Types include all constructs in $\lambda_\mathtt{i}$: a top type $\top$; the type of integers $\mathtt{Int}$; function types $\mathsf{A} \to \mathsf{B}$; and intersection types $\mathsf{A}\&\mathsf{B}$. Types are extended with two standard constructs of System F: type variables α and type abstraction $\forall(\alpha * \mathsf{A}).\, \mathsf{B}$. The latter lifts the System F's quantification into *disjoint quantification*: it includes an extra disjointness constraint tied to a type variable α. Finally, the syntax for the singleton record at type level consists of a label $\mathtt{l}$ and an associated type A.

[2] We do not support exactly this function definition style; however the type signature and expression (module infix operators) are exactly as one would write them in $\mathsf{F}_\mathtt{i}$

$$
\begin{array}{lll}
\text{Types} & A, B & ::= \top \mid \mathtt{Int} \mid A \to B \mid A\&B \\
& & \quad \mid \alpha \mid \forall(\alpha * A).\, B \mid \{l : A\} \\
\text{Terms} & e & ::= \top \mid i \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid e_1 ,, e_2 \\
& & \quad \mid \Lambda(\alpha * A).\, e \mid e\ A \mid \{l = e\} \mid e.l \\
\text{Contexts } \Gamma & & ::= \cdot \mid \Gamma, \alpha * A \mid \Gamma, x{:}A
\end{array}
$$

Fig. 1. F_i syntax.

Terms. Metavariables e range over terms. Terms include all constructs in λ_i: a unit type $()$; an integer literal i; a variable x, abstraction of terms over variables of a given type $\lambda x.\, e$; application of terms e_1 to terms e_2, written $e_1\ e_2$; and the *merge* of terms e_1 and e_2 denoted as $e1 ,, e2$, corresponding to intersections of types $A\&B$. 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. The syntax for the singleton record at term level 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 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.

In order to focus on the key features that make this language interesting, we do not include other forms such as type constants and fixpoints here. However they can be included in the formalization in standard ways and we are using them in discussions and examples.

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 first three rules are rather straightforward: ($S\top$) says that every type is a subtype of $\top$; ($S\mathbb{Z}$) and ($S\alpha$) define subtyping as a reflexive relation on integers and type variables. The rule ($S\to$) says that a function is contravariant in its parameter type and covariant in its return type. The three rules dealing with intersection types are just what one would expect when interpreting types as sets. Under this interpretation, for example, the rule ($S\&R$) says that if A_1 is both the subset of A_2 and the subset of A_3, then A_1 is also the subset of the intersection of A_2 and A_3. The `ordinary` conditions are necessary to ensure coherence [**?**]. We will come back to this in the next section.

In ($S\forall$) a universal quantifier ($\forall$) is covariant in its body, and contravariant in its disjointness constraints. Finally, ($S\textsc{Rec}$) says records are covariant within

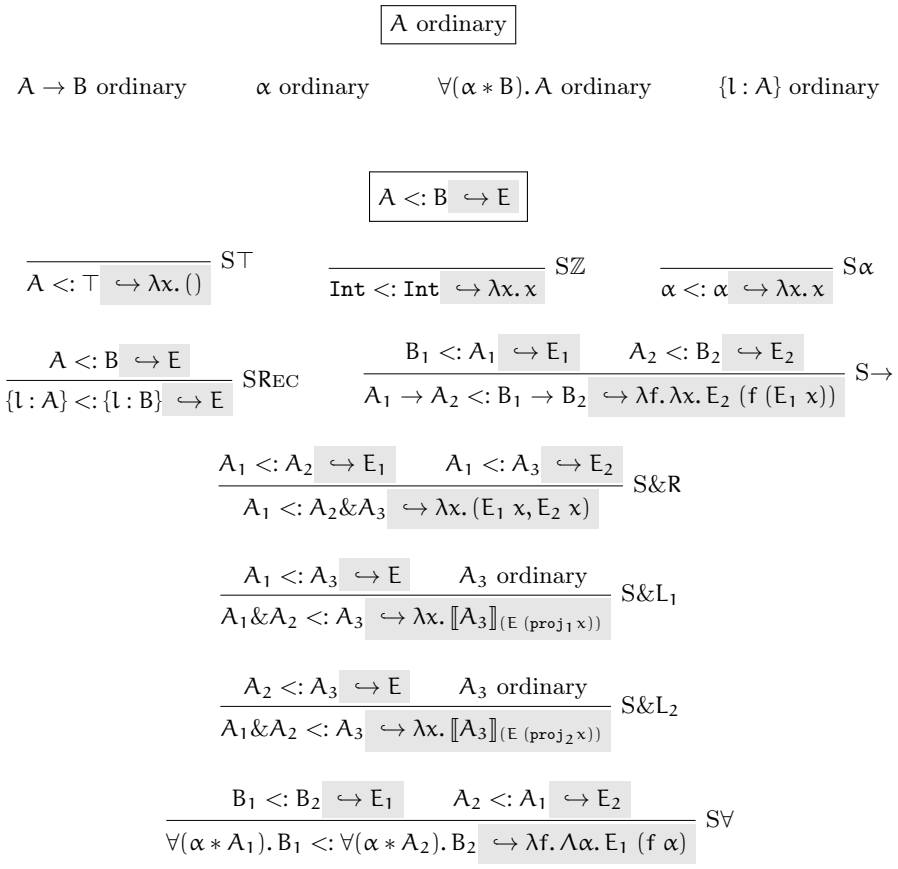

Fig. 2. Subtyping rules of F_i.

their fields' types. Subtyping is reflexive and transitive.BRUNO: State as lemmas here, since this is new.

section?

4.3 Typing

Well-formedness The well-formedness rules are shown in the top part of Figure 4.3. The new rules are (WFα) and (WF$\forall$). Their definition is quite straightforward, but note how we ensure the well-formedness of the constraint in the latter. JOAO: we don't need this

Disjoint quantification. A disjoint quantification is introduced by the big lambda $\Lambda(\alpha * A).\, e$ and eliminated by the usual type application $e\ A$. The constraint is added to the context with this rule. During a type application, the type system

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

$$\frac{}{\Gamma \vdash \top}\ \mathrm{WF}\top \qquad \frac{}{\Gamma \vdash \mathtt{Int}}\ \mathrm{WF}\mathbb{Z} \qquad \frac{\alpha * A \in \Gamma}{\Gamma \vdash \alpha}\ \mathrm{WF}\alpha \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}\ \mathrm{WF}\to$$

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

$$\frac{\Gamma \vdash A}{\Gamma \vdash \{l : A\}}\ \mathrm{WFRec}$$

makes sure that the type argument satisfies the disjointness constraint. We will explain next the rules that compose the type-system.

Typing rules JOAO: we can safely remove the WF A in T-BLam 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. The typing rules are shown in the bottom part of the figure. 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 to type A' 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). The new rules, inspired on the standard System F, are the infer rules for type application (T-TApp), and for type abstraction (T-BLam). The former is no different from the standard rule for type application while the latter has an extra hypothesis ensuring that the type to be instantiated is compatible (i.e. disjoint) with the constraint associated with the abstracted variable. This is extremely important, as it will retain the desired coherence of our type-system. For the 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. 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 that we will prove to be both type-safe and coherent. However,

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

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

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

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

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

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

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

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

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

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

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

Fig. 3. Type system of F_{i}.

before we can prove such properties, it is necessary to introduce our new version of disjointness, considering polymorphism and disjointness quantificaton. This section first presents the set of rules for determining whether two types are disjoint. The set of rules is algorithmic and an implementation is easily derived from them. 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

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

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

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

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

$$\frac{\Gamma \vdash A *_i B_1 \qquad \Gamma \vdash A *_i B_2}{\Gamma \vdash A *_i B_1 \& B_2}\ *\&R \qquad \frac{A *_{ax} B}{\Gamma \vdash A *_i B}\ *Ax$$

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

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

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

Fig. 4. Algorithmic Disjointness.

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

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. $(*\top)$ and $(*\top\mathtt{Sym})$ say that any type is disjoint to $\top$. $(*\alpha)$ is the base rule and $(*\alpha\mathtt{Sym})$ is its twin rule BRUNO: I think we should change the names of the rules. Perhaps follow the subtyping relation and use the letter "D". Say $D_\top$ for the disjointness rule for top.. JOAO: added lines on top of axioms, this already cleans up a bit. 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:

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

The lemma states that if a type A is disjoint to B under Γ, then it is also disjoint to any supertype of B. The rule for disjoint quantification ($*\forall$) 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 ($*\rightarrow$) says that two function types are disjoint if and only if their return types are disjoint. The rules dealing with intersection types (($*\&$L) and ($*\&$R)) say that an intersection is disjoint to some type B, whenever both of their components are also disjoint to B. Finally, the rule ($*$Ax) 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. Just as in ..., 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. Note that disjointness with the $\top$ type is already captured by ($*\top$) and ($*\top\texttt{Sym}$).

5.2 Substitution metatheory

Disjointness will not only play a fundamental role in ensuring coherence, but also in ensuring the type-safety of our system. Since the type-system is only allowed to instantiate a type variable with other types which are disjoint to the variable's disjointness constraint, one might ask: what are the exact implications of mixing substitution with disjoint intersection types? We will next dive into this question in greater detail.

Disjoint substitutions One rule of thumb in disjoint intersection types states 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 won't 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 `Int` are disjoint:

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

After the substitution of `Int` for α on the two types, the sentence

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

is no longer true since `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.

Now, more formally, we can show following lemma holds:

JOAO: missing WFEnv premisses

Lemma 1 (Disjointness is stable under substitution).
If $(\mathsf{x} * \mathsf{C}) \in \Gamma$ *and* $\Gamma \vdash \mathsf{C} * \mathsf{D}$*, then* $\Gamma[\mathsf{x} := \mathsf{C}] \ \vdash \mathsf{A}[\mathsf{x} := \mathsf{C}] \ * \mathsf{B}[\mathsf{x} := \mathsf{C}]$ *,*

where $\Gamma[\mathsf{x} := \mathsf{C}]$ means substituting x by C in the co-domain of the environment.

Proof. By induction on the disjointness derivation of C and D. Special attention for the variable case, where it was necessary to prove stability of substitution for the subtyping relation. It was 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:

JOAO: again, missing WFEnv premises

Lemma 2 (Types are stable under substitution).
If $\Gamma \vdash \mathsf{A}$ *and* $\Gamma \vdash \mathsf{B}$ *and* $(\mathsf{x} * \mathsf{C}) \in \Gamma$ *and* $\Gamma \vdash \mathsf{B} * \mathsf{C}$*, then* $\Gamma[\mathsf{B} := \mathsf{x}] \ \vdash \mathsf{A}[\mathsf{B} := \mathsf{x}]$ *.*

Proof. By induction on the well-formedness derivation of A. The intersection case requires the use of Lemma 1. 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 3 (Well-formed typing).
If $\Gamma \vdash \mathsf{e} \Leftarrow \mathsf{A}$*, then* $\Gamma \vdash \mathsf{A}$*.*
If $\Gamma \vdash \mathsf{e} \Rightarrow \mathsf{A}$*, then* $\Gamma \vdash \mathsf{A}$*.*

Proof. By induction on the derivation and applying Lemma 2 in the case of (T-TApp).

Even though the meta-theory is consistent with the expected results, there is still an open 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 expected, 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, let us restate a previous lemma, which we used to justify the rule for disjointness of variables:

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

If one takes A as some variable x, and B as some type in the environment, we can interpreting how many possible choices are there for the type C. Given that the cardinality of types is infinite, we will strict C to only 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 immediately conclude that 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, more choices for C means that there will be less choices to instantiate the variable x. Thus the options for instantiating x are inversely proportional to the number of intersections present in B. There are two special cases, namely $\top$ (i.e. the 0-ary intersection) and $\bot$ (i.e. the infinite intersection) [3]. We will discuss them next.

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

The less restrictive bound Inversely, the most restrictive type should be $\bot$, as one might think of $\bot$ as specific as the infinite intersection. In other words, $\bot$ is not disjoint to any type, except top-like types. However, introducing $\bot$ is not compatible with our disjointness rule $(*\alpha)$ and well-formedness of contexts. Let

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

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

Fig. 5. Target language syntax.

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. For this end, we want to use $(*\alpha)$ 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 are now able to derive it, as A can now be $\bot$: a valid sub-type of x which does not contain x. In fact, introducing any *bottom-like* type (i.e. $\bot \& B$) can lead to this undesired behaviour. Since defining the lower bound is not strictly necessary to our formalization; introduces substantial complexity in our system; and its practical application is still not clear, we left this as an open problem for future work.

6 Semantics and Coherence

BRUNO: You are repeatedily referring to our previous paper and Dunfield's paper. You don't need to constantly remind the reader of this. Remove some of this repetition.

This section discusses the elaboration semantics of F_i and show how coherence is retained. We will first explain the semantics by means of the elaboration to System F. Then, we 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

We define the dynamic semantics of the call-by-value F_i by means of a type-directed translation to an extension of System F with pairs [4].

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 the appendix.

Type and context translation. Figure 6 defines the type translation function $|\cdot|$ from F_i types A to target language types T. The notation $|\cdot|$ is also overloaded for context translation from F_i contexts Γ to target language contexts G.

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

$\boxed{|A| = T}$

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

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

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

Fig. 6. Type and context translation.

$\boxed{\rceil A \lceil}$

$$\frac{}{\rceil \top \lceil}\ \textsc{Toplike-Top} \qquad \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \& B \lceil}\ \textsc{Toplike-Inter} \qquad \frac{\rceil B \lceil}{\rceil A \to B \lceil}\ \textsc{Toplike-Fun}$$

$$\frac{\rceil A \lceil}{\rceil \forall(\alpha * B).\, A \lceil}\ \textsc{Toplike-Forall}$$

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

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

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

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

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

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 similarity of these rules with the simply-typed system, our notion of disjointness has changed. Consequently, the set of well-formed top-like types has changed and we also adjusted the meta-function $[\![A]\!]$, as shown in the bottom of Figure 7. Note how not only the $\forall$ case is defined, but also the intersection case. This is extremely important as it plays a fundamental role in ensuring the coherence of subtyping, as we will describe next.

6.3 Coercive Subtyping and Coherence

Coercive subtyping. The judgement

$$A_1 <: A_2 \hookrightarrow E$$

extends the subtyping judgement in Figure 2 with a coercion on the right hand side of $\hookrightarrow$. A coercion E is just an term in the target language and is ensured to have type $|A_1| \to |A_2|$ (by Lemma 4). For example,

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

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

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 4 (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 intersections the following theorem holds:

Lemma 5 (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. $A_2 <: B$

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

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. BRUNO: Don't just copy&paste from the previous paper. You should mention the new rules, which Dunfield does not have. This translation turns merges into usual pairs, similar to Dunfield's elaboration approach [18]. For example,

$$1,,\texttt{"one"}$$

becomes $(1, \texttt{"one"})$. In usage, the pair will be coerced according to type information. For example, consider the function application:

$$(\lambda x.\, x)\ (1,,\texttt{"one"})$$

This expression will be translated to

$$(\lambda x.\, x)\ ((\lambda x.\, \texttt{proj}_2 x)\ (1, \texttt{"one"}))$$

The coercion in this case is $(\lambda x.\, \texttt{proj}_2 x)$. It extracts the second item from the pair, since the function expects a `String` but the translated argument is of type $(\texttt{Int}, \texttt{String})$.

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 might also be noteworthy saying 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. (Sketch) 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* $A_1 = A_2$.

JOAO: review proof

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

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

JOAO: review this proof 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.

Proof. Note that the typing rules are already syntax-directed but the case of (T-App) (copied below) still needs special attention since we need to show that the generated coercion E is unique.

$$\frac{\Gamma \vdash e_1 : A_1 \to A_2 \hookrightarrow E_1 \qquad \Gamma \vdash e_2 : A_3 \hookrightarrow E_2 \qquad A_3 <: A_1 \hookrightarrow E}{\Gamma \vdash e_1\ e_2 : A_2 \hookrightarrow E_1\ (E\ E_2)}\ \text{T-App}$$

Luckily, by Lemma 3, we know that typing judgements give well-formed types, and thus $\Gamma \vdash A_1$ and $\Gamma \vdash A_3$. Therefore we are able to apply Lemma 6 and conclude that E is unique.

7 Related Work

Coherence Reynolds invented Forsythe [30] in the 1980s. Our merge operator is analogous to his operator $\mathtt{p}_1, \mathtt{p}_2$. Forsythe has a coherent semantics. The result was proved formally by Reynolds [29] 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 $\mathsf{F}^*_{\mathsf{i}}$ 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 2.4 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 $\mathsf{F}^*_{\mathsf{i}}$ 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, $\mathsf{F}^\omega_\wedge$, to model mul-

tiple 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 `Int` as $\alpha \setminus \texttt{Int}$, which is syntactic sugar for $\alpha \wedge \neg \texttt{Int}$. As a comparison, such a type will need a disjoint quantifier, like $\forall(\alpha * \texttt{Int}).\,\alpha$, 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 $\mathsf{F}_{\mathsf{i}}^{*}$ 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 [36]. 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 [37] 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 [31,24]. 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 [30] 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 $\mathsf{F}_{<:\rho}$, 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 [38] started the work on extensible records and proposed row types [39] 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 $\mathsf{F}^*_{\mathsf{i}}$: 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 $\mathsf{F}^*_{\mathsf{i}}$ as part of a JVM-based compiler. Based on the type system of $\mathsf{F}^*_{\mathsf{i}}$, 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 $\mathsf{A}|\mathsf{B}$, which is the common supertype of two types A and B.

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. 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)
30. Reynolds, J.C.: Design of the programming language Forsythe (1997)
31. 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)
32. 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)
33. 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)
34. Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423–436 (July 2008)
35. 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)
36. Wadler, P.: The expression problem. Java-genericity mailing list (1998)
37. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: POPL. pp. 60–76. ACM (1989)
38. Wand, M.: Complete type inference for simple objects. In: LICS (1987)
39. 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)
40. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: FOOL (2005)