Disjoint Polymorphism

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

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

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

1 Introduction

Intersection types [18,40] are a popular language feature for modern languages, such as Microsoft's TypeScript [4], Redhat's Ceylon [1], Facebook's Flow [3] and Scala [34]. In those languages a typical use of intersection types, which has been known for a long time [17], is to model the subtyping aspects of OO-style multiple inheritance. For example, the following Scala declaration:

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

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

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

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

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

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

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

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

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

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

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

The *extend* function is essentially an encoding of the *merge operator*. The merge operator is used on some calculi [45,44,15,22,35] as an introduction form for intersection types. Similar encodings to those in TypeScript have been proposed for Scala to enable applications where the merge operator also plays a fundamental role [36,43]. Unfortunately, the merge operator is not directly supported by TypeScript, Scala, Ceylon or Flow. There are two possible reasons for such lack of support. One reason is simply that the merge operator is not well-known: many calculi with intersection types in the literature do not have explicit introduction forms for intersection types. The other reason is that, while powerful, the merge operator is known to introduce *(in)coherence* problems [44,22]. If care is not taken, certain programs using the merge operator do not have a unique semantics, which significantly complicates reasoning about programs.

Solutions to the problem of coherence in the presence of a merge operator exist for simply typed calculi [45,44,15,35], but no prior work addresses polymorphism. Most recently, we proposed using *disjoint intersection types* [35] to guarantee coherence in λ_i: a simply typed calculus with intersection types and a merge operator. The key idea is to allow only disjoint types in intersections. If two types are disjoint then there is no ambiguity in selecting a value of the appropriate type from an intersection, guaranteeing coherence.

Combining parametric polymorphism with disjoint intersection types, while retaining enough flexibility for practical applications, is non-trivial. The key issue

is that when a type variable occurs in an intersection type it is not statically known whether the instantiated types will be disjoint to other components of the intersection. A naive way to add polymorphism is to forbid type variables in intersections, since they may be instantiated with a type which is not disjoint to other types in an intersection. Unfortunately this is too conservative and prevents many useful programs, including the `extend` function, which uses an intersection of two type variables `T` and `U`.

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

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

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

The applicability of F_i is illustrated with examples using `extend` ported from TypeScript, and various operations on polymorphic extensible records [31,27,29]. The operations on polymorphic extensible records show that F_i can encode various operations of row types [49]. However, in contrast to various existing proposals for row types and extensible records, F_i supports general intersections and not just record operations.

F_i and the proofs of coherence and type-safety are formalized in the Coq theorem prover [2]. The proofs are complete except for a minor (and trivially true) variable renaming lemma used to prove the soundness between two subtyping relations used in the formalization. The problem arizes from the combination of the locally nameless representation of binding [7] and existential quantification, which prevents a Coq proof for that lemma.

In summary, the contributions of this paper are:

- **Disjoint Polymorphism:** A novel form of universal quantification where type variables can have disjointness constraints. Disjoint polymorphism enables a flexible combination of intersection types, the merge operator and parametric polymorphism.
- **Coq Formalization of F_i and Proof of Coherence:** An elaboration semantics of System F_i into System F is given. Type-soundness and coherence are proved in Coq. The proofs for these properties and all other lemmata found in this paper are available at:
  `https://github.com/jalpuim/disjoint-polymorphism`
- **Applications:** We show how F_i provides basic support for dynamic mixins and various operations on polymorphic extensible records.

2 Overview

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

2.1 Intersection Types and the Merge Operator

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

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

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

Merge operator. The previous program deliberately omitted 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* [45,44,15,22,35]. As Dunfield [22] has argued a merge operator adds considerable expressiveness to a calculus. The merge operator allows two values to be merged in a single intersection type. For example, an implementation of `x` in F_i is `1,,True`. Following Dunfield's notation the merge of v_1 and v_2 is denoted by $v_1,,v_2$.

2.2 Coherence and Disjointness

Coherence is a desirable property for a semantics. A semantics is coherent if any *valid program* has exactly one meaning [44] (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`. The result of the program `(1,,2) : Int`) can be either `1` or `2`, depending on the implementation of the language.

Disjoint intersection types One option to restore coherence is to reject programs which may have multiple meanings. The λ_i calculus [35] – a simply-typed calculus with intersection types and a merge operator – solves this problem by using the concept of disjoint intersections. The incoherence problem with the expression $1,,2$ happens because there are two overlapping integers in the merge. Generally speaking, if both terms can be assigned some type C then both of them can be

chosen as the meaning of the merge, which in its turn leads to multiple meanings of a term. Thus a natural option is to forbid such overlapping values of the same type in a merge. In λ_i intersections such as `Int&Int` are forbidden, since the types in the intersection overlap (i.e. they are not disjoint). However an intersection such as `Char&Int` is OK because the set of characters and integers are disjoint to each other.

2.3 Parametric Polymorphism

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

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

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

```
merge3 Int 2
```

can evaluate to both 2 or 3.

Forbidding type variables in intersections A naive way to ensure that only programs with disjoint types are accepted is simply to forbid type variables in intersections. That is, an intersection type such as `Char&Int` would be accepted, but an intersection such as `A&Int` (where `A` is some type variable) would be rejected. The reasoning behind this design is that type variables can be instantiated to any types, including those already in the intersection. Thus forbidding type variables in the intersection will prevent invalid intersections arising from instantiations with overlapping types. Such design does guarantee coherence and would prevent `merge3` from type-checking. Unfortunately the big drawback is that the design is too conservative and many other (useful) programs would be rejected. In particular, the `extend` function from Section 1 would also be rejected.

Other approaches Another option to mitigate the issues of incoherence, without the use of disjoint intersection types, is to allow for a biased choice: multiple values of the same type may exist in an intersection, but an implementation gives preference to one of them. The encodings of merge operators in TypeScript and Scala [36,43] use such an approach. A first problem with this approach, which has already been pointed out by Dunfield [22], is that the choice of the corresponding value is tied up to a particular choice in the implementation. In other words incoherence still exists at the semantic level, but the implementation makes it predictable which overlapping value will be chosen. From the theoretical point-of-view it would be much better to have a clear, coherent semantics, which is independent from concrete implementations. Another problem is that the interaction between biased choice and polymorphism can lead to counter-intuitive programs, since instantiation of type-variables affects the type-directed lookup of a value in an intersection.

2.4 Disjoint Polymorphism

To avoid being overly conservative, while still retaining coherence in the presence of parametric polymorphism and intersection types, F_i uses *disjoint polymorphism*. Inspired by bounded quantification [13], where a type variable is constrained by a type bound, disjoint polymorphism allows type variables to be constrained so that they are disjoint to some given types.

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

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

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

```
merge3 Bool True
```

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

Multiple constraints Disjoint quantification allows multiple constraints. For example, the following variant of `merge3` has an additional boolean in the merge:

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

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

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

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

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

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

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

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

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

Empty constraint The type variable A in the `fst` function has no constraint. In F_i this actually means that A should be associated with the empty constraint, which raises the question: which type should be used to represent such empty constraint? Or, in other words, which type is disjoint to every other type? It is obvious that this type should be one of the bounds of the subtyping lattice: either $\bot$ or $\top$. The essential intuition here is that the more specific a type in the subtyping relation is, the less types exist that are disjoint to it. For example, `Int` is disjoint to all types except the intersections that contain `Int`, `Int` itself, and $\bot$; while `Int`&`Char` is disjoint to all types that do not contain `Int` or `Char`, and $\bot$. 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 $A * \top$.

2.5 Stability of Substitutions

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

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

In F_i if a type variable A is substituted in a type T_1, for a type T_2 (written $[A := T_2]\ T_1$), 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 $\forall(B * \texttt{Int}).(\texttt{Int}\&B) \to \texttt{Int}$. Now, the second `Int` application is problematic, since instantiating B with `Int` will lead to the ill-formed type (`Int`&`Int`) $\to$ `Int`. However, from this example it is easy to see that all types which are not problematic are exactly the ones disjoint with A. This paper shows how a weaker version of the usual type substitution stability still holds, namely by requiring that the type variable's disjointness constraint is compatible with the type as target of the instantiation.

3 Applications

F_i is illustrated with two applications. The first application shows how to mimic some of TypeScript's examples of dynamic mixins in F_i. The second application shows how F_i enables a powerful form of polymorphic extensible records.

3.1 Dynamic Mixins

TypeScript is a language that adds static type checking to JavaScript. Amongst numerous static typing constructs, *TypeScript supports a form of intersection types, without a merge operator*. However, it is possible to define a function `extend` that mimics the merge operator:

```
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",true), new ConsoleLogger());
var n = jim.name;
jim.log();
```

In this example, taken from TypeScript's documentation[2], an `extend` function is defined for mixin composition. Two classes `Person` and `ConsoleLogger` are also defined. Two instances of those classes are then composed in a variable `jim` with the type of the intersection of both using `extend`. It is type-safe to access both the properties from `Person` and `ConsoleLogger` in the object `jim`.

TypeScript's implementation of `extend` relies on a biased choice. The function starts by creating a variable `result` with the type of the intersection. It then iterates through `first`'s properties and copies them to `result`. Next, it iterates through `second`'s properties but it only copies the properties that `result` does not possess yet (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 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 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","yes"),new
    Person("Arnold",true));
boolean b = fool.male;      /* Undetected type-error here! */
```

There are a few problems here. Firstly both `Dog` and `Person` contain a `name` field, and the use of `extend` will favour the `name` field in the first object. This could be surprising for someone unfamiliar with the semantics of `extend` and, more

[2] We have added the field `male` to the class `Person`.

importantly, it could easily allow unintended name clashes to go undetected. Secondly, note how `fool.male` is statically bound to a variable of type `boolean` but, at run-time, it will contain a value of type `String`! Thus the example shows some run-time type errors can still occur when using `extend`.

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 for the practical application of intersection types.

Dynamic mixins in $\mathsf{F_i}$ In $\mathsf{F_i}$, the merge operator is built-in. Thus `extend` is simply defined as follows:

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

The disjointness constraint on `U` ensures that no conflicts (such as duplicated fields of the same type) exists when merging the two objects. In practice this approach is quite similar to trait-based OO approaches [47]. If conflicts exist when two objects are composed, then they have to be resolved manually (by dropping fields from some object, for example). Moreover if no existing implementation can be directly reused, a new one must be provided via record extension, analogously to standard method overriding in OO languages.

For the previous TypeScript examples, assuming a straightforward translation from objects to (polymorphic) records, then the composition of `person` and `consoleLogger` is well-typed in $\mathsf{F_i}$:

```
type Person = {name : String} & {male : Bool};
type Loggable = {log : ⊤ → ⊤};

let person (n : String) (s : Bool) : Person = {name = n} ,, {male = s} in
let consoleLogger : Loggable = {log = ...} in
let jim = extend Person Loggable (person "Jim" true)  consoleLogger in
let n = jim.name in
jim.log ⊤
```

However, the intersection `Dog & Person` is not accepted. This is due to both types sharing a field with the same name (`name`) and the same type (`String`). Note that the name clash between `male` fields (which have different types) does not impose any problem in this example: $\mathsf{F_i}$ allows and keeps duplicated fields whose types are disjoint. This feature of $\mathsf{F_i}$ is further illustrated next.

3.2 Extensible Records

$\mathsf{F_i}$ can encode polymorphic extensible records. Describing and implementing records within programming languages is certainly not novel and has been extensively studied in the past, including systems with row types [49,50]; predicates [28,27,26]; flags [42]; conditional constraints [39]; cases [10]; amongst others. However, while most systems have non-trivial built-in constructs to model various aspects of records, $\mathsf{F_i}$ specializes the more general notion of intersection types to encode complex records.

Records and record operations in $\mathsf{F}_\mathtt{i}$ Systems with records usually rely on 3 basic operations: selection, restriction and extension/concatenation. Selection and concatenation (via the merge operator) are built-in in the semantics of $\mathsf{F}_\mathtt{i}$. Merges in $\mathsf{F}_\mathtt{i}$ can be viewed as a generalization of record concatenation. In $\mathsf{F}_\mathtt{i}$, following well-known encodings of multi-field records in systems with intersection types and a merge operator [45,44], there are only three rather simple constructs for records: 1) single field record types; 2) single field records; 3) field accessors. Multi-field records in $\mathsf{F}_\mathtt{i}$ are encoded with intersections and merges of single field records. An example is already illustrated in Section 3.1. The record type `Person` is the intersection of two single field record types. The record `person "Jim" true` is built with a merge of two single field records. Finally, `jim.name` and `jim.log` illustrates the use of field accessors. Note how, through the use of subtyping, accessors will accept any intersection type that contains the single record with the corresponding field. This resembles systems with record subtyping [14,38].

Restriction via subtyping In contrast to most record systems, restriction is not directly embedded in $\mathsf{F}_\mathtt{i}$. Instead, $\mathsf{F}_\mathtt{i}$ uses subtyping for restriction:

```
let remove (x : {age : Int} & {name : String}) : {name : String} = x in ...
```

The function `remove` drops the field `age` from the record `x`.

Polymorphic extensible records Records in $\mathsf{F}_\mathtt{i}$ can have polymorphic fields, and disjointness enables encoding various operations expressible in systems with polymorphic records. For example, the following variant of `remove`

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

takes a value `x` which contains a record of type `l : A`, as well as some extra information of type `B`. The disjointness constraint on `B` ensures that values of type `B` do not contain a record with type `l : A`. This example shows that one can use disjoint quantification to express negative field information, which is very close to the system described by Harper and Pierce [27]. Note, however, that $\mathsf{F}_\mathtt{i}$ requires explicitly stating the type of field in the constraint, whereas systems with a *lacks* (field) predicate only require the name of the field. The generality of disjoint intersection types, which allows one to encode record types, is exactly what forces us to add this extra type in the constraint. However, there is a slight gain with $\mathsf{F}_\mathtt{i}$'s approach: remove allows `B` to contain fields with label `l`, as long as the field types are *disjoint* to A. Such fine-grained constraint is not possible to express only with a lacks predicate.

Expressibility As noted by Leijen [31], systems can typically be categorized into two distinct groups in what concerns extension: strict and 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.

With *lightweight extensible records* [29] – a system with strict extension – an example of a function that uses record types is the following:

```
let avg₁ (R\x, R\y) => (r : {R | x:Int, y:Int}) = (r.x+r.y)/2
```

The type signature says that any record r, containing fields x and y and some more information R (which lacks both fields x and y), can be accepted 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 Leijen's [31] system with free extension the more general program would be accepted:

```
let avg2 R (r : {x:Int, y:Int | R}) = (r.x+r.y)/2
```

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, in a record with multiple x fields, the most recent (i.e. left-most) is used in any function which accesses x.

In F_i the following program can written instead:

```
let avg3 (R*{x:Int}&{y:Int}) (r : {x:Int}&{y:Int}&R) = (r.x+r.y)/2
```

Since F_i accepts duplicated fields as long as the types of the overlapping fields are disjoint, more inputs are accepted by this function than in the first system. However, since Leijen's system accepts duplicated fields even when types are overlapping, avg_3 accepts less types than avg_2. Another major difference between F_i and the two other mentioned systems, is the ability to combine records with arbitrary types. Our system does not account for well-formedness of record types as the other two systems do (i.e. using a special *row* kind), since our encoding of records piggybacks on the more general notion of disjoint intersection types.

4 The F_i Calculus

This section presents the syntax, subtyping, and typing of F_i: a calculus with intersection types, parametric polymorphism, records and a merge operator. This calculus is an extension of the λ_i calculus [35], which is itself inspired by Dunfield's calculus [22]. F_i extends λ_i with (disjoint) polymorphism. Section 5 introduces the necessary changes to the definition of disjointness presented by Oliveira et al. [35] in order to add disjoint polymorphism.

4.1 Syntax

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

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

Types. Metavariables A, B range over types. Types include all constructs in λ_i: a top type $\top$; the type of integers $\mathtt{Int}$; 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 associated 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 $ftv(\cdot)$ 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 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 all disjointness constraints). Throughout this paper, we will assume that all contexts are well-formed. Importantly, besides usual well-formedness conditions, in well-formed contexts type variables must not appear free within its own disjointness constraint.

Syntactic sugar In F_i we may quantify a type variable and omit its constraint. This means that its constraint is $\top$. For example, the function type $\forall\alpha.\alpha \to \alpha$ is syntactic sugar for $\forall(\alpha * \top).\, \alpha \to \alpha$. This is discussed in more detail in Section 6.

4.2 Subtyping

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

Polymorphism and Records. The subtyping rules introduced by F_i refer to polymorphic constructs and records. Sα defines subtyping as a reflexive relation on type variables. In S$\forall$ a universal quantifier ($\forall$) is covariant in its body, and contravariant in its disjointness constraints. The SRec rule says that records are covariant within their fields' types. The subtyping relation uses an auxiliary unary `ordinary` relation, which identifies types that are not intersections. The `ordinary` conditions on two of the intersection rules are necessary to produce unique coercions [35]. The `ordinary` relation needs to be extended with respect to λ_i. As shown at the top of Figure 1, the new types it contains are type variables, universal quantifiers and record types.

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

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

Proof. By induction on A. □

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

Proof. By double induction on both derivations. □

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

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

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

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

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

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

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

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

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

Fig. 1. Subtyping rules of F_i.

4.3 Typing

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

Typing rules. Our typing rules are formulated as a bi-directional type-system. Just as in λ_i, this ensures the type-system is not only syntax-directed, but also that there is no type ambiguity: that is, inferred types are unique. The typing rules are shown in the bottom part of Figure 2. Again, the reader is advised to ignore the gray-shaded parts, as these 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 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Well-formedness and type system of F_i.

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; and it will be further explained in Section 5. 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 straightforwardly 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 inferring type A.

5 Disjointness

Section 4 presented a type system with disjoint intersection types and disjoint quantification. In order to prove both type-safety and coherence (in Section 6), 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 F_i.

5.1 Algorithmic Rules for Disjointness

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

Main judgement. The judgement $\Gamma \vdash A * B$ says two types A and B are disjoint in a context Γ. The rules are inspired in the disjointness algorithm described by λ_i. D$\top$ and D$\top$Sym say that any type is disjoint to $\top$. This is a major difference to λ_i, where the notion of disjointness explicitly forbids the presence of $\top$ types in intersections. We will further discuss this difference in Section 6.

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

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

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

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

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

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

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

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

$$\frac{B *_{ax} A}{A *_{ax} B}\ \mathrm{D}_{\mathrm{Ax}}\mathrm{Sym} \qquad \frac{}{\mathtt{Int} *_{ax} A_1 \to A_2}\ \mathrm{D}_{\mathrm{Ax}}(\mathbb{Z}\to) \qquad \frac{}{\mathtt{Int} *_{ax} \{l = A\}}\ \mathrm{D}_{\mathrm{Ax}}(\mathbb{Z}\mathrm{Rec})$$

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

$$\frac{}{A_1 \to A_2 *_{ax} \{l = B\}}\ \mathrm{D}_{\mathrm{Ax}}(\to\mathrm{Rec}) \qquad \frac{}{\forall(\alpha * A_1).\, A_2 *_{ax} \{l = B\}}\ \mathrm{D}_{\mathrm{Ax}}(\forall\mathrm{Rec})$$

Fig. 3. Algorithmic disjointness.

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

The lemma states that if a type A is disjoint to B under Γ, then it is also disjoint to any supertype of B. Note how these two variable rules would allow one to prove $\alpha * \alpha$, for any variable α. However, under the assumption that contexts are well-formed, such derivation is not possible as α cannot occur free in A.

The rule for disjoint quantification D$\forall$ is the last novel rule. To illustrate this rule, consider the following two types:

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

When are these 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 adds a constraint composed of the intersection of both constraints into Γ and checks for disjointness in the bodies under that environment. 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. This is not problematic because the intersections present

as constraints in the environment do not contribute directly to the (coherent) coercions generated by the type-system. In other words, intersections play two different roles in $\mathsf{F_i}$:

1. As types: restricted (i.e. disjoint) intersections are required to ensure coherence.
2. As constraints: unrestricted intersections are sufficient for constraints under polymorphic instantiation.

The remaining rules are identical to the original rules.

Axioms. Axiom rules take care of two types with different language constructs. These rules capture the set of rules is that $\mathsf{A} *_{\text{ax}} \mathsf{B}$ 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 $\mathrm{D}\alpha$ and $\mathrm{D}\alpha\mathtt{Sym}$, and disjointness with the $\top$ type is captured by $\mathrm{D}\top$ and $\mathrm{D}\top\mathtt{Sym}$.

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 $\forall(\mathsf{A} * \mathtt{Int}).\,(\mathtt{Int}\&\mathsf{A}) \to \mathtt{Int}$, the type A cannot be substituted by any type. However, under certain conditions, weaker versions of substitution lemmas do hold. The conditions are guaranteed by the type-system by only allowing instantiation 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 $\mathsf{A}\&\mathsf{B}$ 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 $\mathsf{A}\&\mathsf{B}$, or, more generally, disjointness should be stable under substitution. Let us illustrate this with an example. 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. Generally speaking, a careless substitution can violate the disjoint constraint in the context. This explains the need to ensure that during type-instantiation the target of the substitution is compatible with such disjointness constraint.

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

Lemma 5 (Disjointness is stable under substitution). *If* $(\alpha * \mathsf{D}) \in \Gamma$ *and* $\Gamma \vdash \mathsf{C} * \mathsf{D}$ *and* $\Gamma \vdash \mathsf{A} * \mathsf{B}$ *and well-formed context* $[\alpha := \mathsf{C}]\ \Gamma$*, then* $[\alpha := \mathsf{C}]\ \Gamma \vdash [\alpha := \mathsf{C}]\ \mathsf{A} * [\alpha := \mathsf{C}]\ \mathsf{B}$.

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 show that their types are stable under substitution, in order to avoid ill-formed types. In the presence of disjoint quantification, we cannot prove such property. However, a weaker version of that property – but strong enough for our type-system's metatheory – can be proven, namely:

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

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

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

Lemma 7 (Well-formed typing). *We have that:*

- *If* $\Gamma \vdash e \Leftarrow \mathsf{A}$*, then* $\Gamma \vdash \mathsf{A}$.
- *If* $\Gamma \vdash e \Rightarrow \mathsf{A}$*, then* $\Gamma \vdash \mathsf{A}$.

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

Even though the meta-theory is consistent, we can still ask: what are the bounds of disjoint quantification? In other words, which type(s) can be used to allow unrestricted instantiation, and which one(s) will completely restrict instantiation? As the reader might expect, the answer is tightly related to subtyping.

5.3 Bounds of Disjoint Quantification

Substitution raises the question of what range of types can be instantiated for a given variable α, under a given context Γ. To answer this question, we ask the reader to recall the rule Dα, copied below:

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

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 $\bot$. Now we may ask: how many suitable types are there to instantiate α with, depending on A? The rule above tells us that the more super-types A has, the more types α has to be disjoint with. In other words, the choices for instantiating α are inversely proportional to the number of super-types of A. It is easy to see that the number of super-types of A is directly proportional to the number of intersections in A. For example, taking A as `Int` leads B to be either $\top$ or `Int`; whereas A as `Int&String` leaves B as either $\top$, `Int` or `String`. Thus, the choices of α are inversely proportional to the number of intersections in A. By analogy, one may think of a disjointness constraint as a set of (forbidden) types, where primitive types are the singleton set and each & is the set union.

Following the same logic, we may conclude that having $\top$ (i.e. the 0-ary intersection) as constraint leaves α with the most options for instantiation; whereas $\bot$ (i.e. the infinite intersection) will deliver the least options. Thus $\top$ is the empty constraint: a variable associated to it can be instantiated to any well-formed type. It is a subtle but very important property, since F_i is a generalization of System F. Any type variable quantified in System F, can be quantified equivalently in F_i by assigning it a $\top$ disjointness constraint (as seen in Section 2.4).

6 Semantics, Coherence and Type-Safety

This section discusses the elaboration semantics of F_i and proves type-safety and coherence. It will first show how the semantics is described by an 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 Target Language

The dynamic semantics of the call-by-value F_i is defined by means of a type-directed translation to an extension of System F with pairs. The syntax and typing of our target language is unsurprising:

$$
\begin{array}{lll}
\text{Types} & T & ::= \alpha \mid \texttt{Int} \mid T_1 \to T_2 \mid \forall \alpha.\, T \mid \boxed{()} \mid \boxed{(T_1, T_2)} \\
\text{Terms} & E & ::= x \mid i \mid \lambda x.\, E \mid E_1\ E_2 \mid \Lambda \alpha.\, E \mid E\ T \mid \boxed{()} \mid \boxed{(E_1, E_2)} \mid \boxed{\texttt{proj}_1\, E} \mid \boxed{\texttt{proj}_2\, E} \\
\text{Contexts} & G & ::= \cdot \mid G, \alpha \mid G, x{:}T
\end{array}
$$

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

Type and context translation. Figure 4 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.

$\boxed{|A| = T}$

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

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

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

Fig. 4. Type and context translation.

6.2 Coercive Subtyping and Coherence

Coercive subtyping. The judgement

$$A_1 <: A_2 \hookrightarrow E$$

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

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

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

Rules S$\top$, Sα, S$\mathbb{Z}$, S$\to$, S&L$_1$, S&L$_2$, and S&R are taken directly from λ_i. In rule Sα, the coercion is simply the identity function. Rule S$\forall$ elaborates disjoint quantification, reusing only the coercion of subtyping between the bodies of both types. Rule SRec elaborates records by simply reusing the coercion generated between the inner types. Finally, all rules produce type-correct coercions:

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

Proof. By a straightforward induction on the derivation. □

Unique coercions In order to prove the type-system coherent, the subtyping relation also needs to be shown coherent. In F_i the following lemma holds:

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

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

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

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

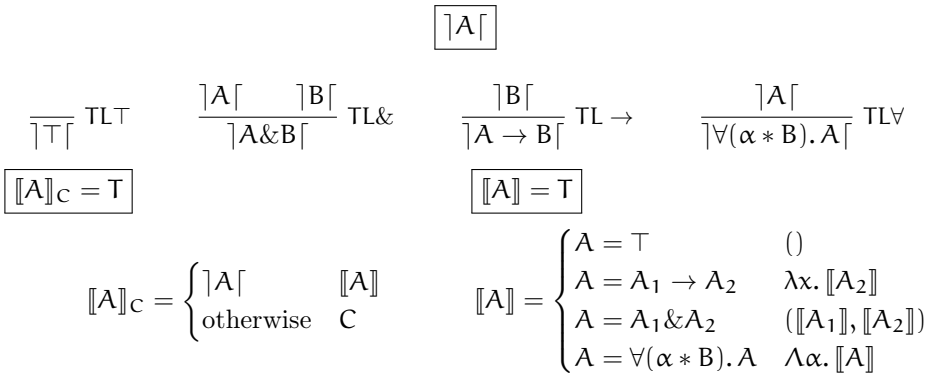

Fig. 5. Top-like types and their coercions.

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

Proof. By induction on the first derivation and case analysis on the second. □

6.3 Top-like Types and their Coercions

Lemma 9, which is fundamental in the proof of coherence of subtyping, holds under the condition that B is not a *top-like type*. Top-like types in F_i include $\top$ as well as other syntactically different types that behave as $\top$ (such as $\top\&\top$). It is easy to see that the unique subtyping contributor lemma is invalidated if no restriction on top-like types exists. Since every type is a subtype of $\top$ then, without the restriction, the lemma would never hold.

Rules. F_i's definition of top-like types extends that from λ_i. The rules that compose this unary relation, denoted as $⌉.⌈$, are presented at the top of Figure 5. The only new rule is TL$\forall$, which states that a (disjoint) universal quantifier is top-like whenever its body is also top-like.

Coercions for top-like types Coercions for top-like types require special treatment for retaining coherence. Although Lemma 9 does not hold for top-like types, we can still ensure that that any coercions for top-like types are unique, *even if multiple derivations exist*. The meta-function $[\![A]\!]$, shown at the bottom of Figure 5, defines coercions for top-like types. With respect to λ_i the $\forall$ case is now defined, and there is also a change in the & case (covering types such as $\top\&\top$). With this definition, although two derivations exist for type $\texttt{Char}\&\texttt{Int} <: \top$, they both generate the coercion $\lambda x.()$.

Allowing overlapping top-like types in intersections In F_i $\top\&\top$ is a well-formed disjoint intersection type. This may look odd at first, since all other types cannot appear more than once in an intersection. Indeed, in λ_i, $\top\&\top$ is not well-formed. However, $\top$ types are special in that they have a *unique* canonical top value,

which is translated to the unit value `()` in the target language. In other words a merge of two top types will always return the same value regardless of which component of the merge is chosen. This is different from merges with other types of values, which do not have this property. It turns out that allowing top-types in intersections is fundamental for the $\top$ type to behave correctly as the empty constraint, since the empty constraint should be disjoint to every other type. This explains the more liberal treatment of top types F_{i} when compared to λ_{i}.

6.4 Elaboration of the Type-System and Coherence

In order to prove the coherence result, we refer to the bidirectional type-system introduced in Section 4. 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 [22]. 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 $\Lambda\alpha.\lambda x.(x,1)$.

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 $|A|$. The two rules for type abstraction (T-BLam) and type application (T-TApp) generate the expected corresponding coercions in System F. The coercions generated for T-Rec and T-ProjR simply erase the labels and translate the corresponding underlying term. All the remaining rules are ported from λ_{i}.

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 respective 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). *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 respective inference rule. □

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

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

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

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

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

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

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

7 Related Work

Intersection types, polymorphism and the merge operator. To our knowledge no previous work presents a *coherent* calculus which includes parametric polymorphism, intersection types and a *merge* operator. Only Pierce's $F_\wedge$ [37] supports intersection types, polymorphism and, as an extension, the merge operator (called `glue` in $F_\wedge$). However, with such extension, $F_\wedge$ is incoherent. Various simply typed systems with intersections types and a merge operator have been studied in the past. The merge operator was first introduced by Reynold's in the Forsythe [45] language. The merge operator in Forsythe is coherent [44], but it has various restrictions to ensure coherence. For example Forsythe merges cannot contain more than one function. Many of those restrictions are lifted in F_i. Castagna et al. [15] studied a coherent calculus with a special merge operator that works on functions only. The goal was to model *function overloading*. Unlike Reynold's operator, multiple functions are allowed in merges, but non-functional types are forbidden. More recently, Dunfield [22] formalised a system with intersection types and a merge operator with a type-directed translation to the simply-typed lambda calculus with pairs. Although Dunfield's calculus is incoherent, it was the inspiration for the λ_i calculus [35], which F_i extends.

λ_i solves the coherence problem for a calculus similar to Dunfield's, by requiring that intersection types can only be composed of *disjoint* types. λ_i uses a specification for disjointness, which says that two types are disjoint if they do not share a common supertype. F_i does not use such specification as its adaptation to polymorphic types would require using unification, making the specification arguably more complex than the algorithmic rules (and defeating the purpose of

having a specification). $\mathsf{F_i}$'s notion of disjointness is based on λ_i's more general notion of disjointness concerning $\top$ types, called $\top$-disjointness. $\top$-disjointness states that two types A and B are disjoint if two conditions are satisfied:

1. (not $\rceil\mathsf{A}\lceil$) and (not $\rceil\mathsf{B}\lceil$)
2. $\forall\mathsf{C}$. if $\mathsf{A} <: \mathsf{C}$ and $\mathsf{B} <: \mathsf{C}$ then $\rceil\mathsf{C}\lceil$

A significant difference between the $\mathsf{F_i}$ and λ_i, is that $\top$-disjointness does not allow $\top$ in intersections, while $\mathsf{F_i}$ allows this. In other words, condition (1) is not imposed by $\mathsf{F_i}$. As a consequence, the set of well-formed *top-like* types is a superset of λ_i's. This is covered in greater detail in Section 6.3.

Intersection types and polymorphism, without the merge operator. Recently, Castagna *et al.* [16] studied a coherent calculus that has polymorphism and set-theoretic type connectives (such as intersections, unions, and negations). Their calculus is based on a semantic subtyping relation due to their interpretation of intersection types. The difference to $\mathsf{F_i}$, is that their intersections are used between function types, allowing overloading (i.e. branching) of types. For example, they can express a function whose domain is defined on any type, but executes different code depending on that type:

$$\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\setminus\texttt{Int}\to\alpha\setminus\texttt{Int})}x.(x \in \texttt{Int})?(x \bmod 2) = 0 : x$$

In our system we cannot express some of these intersections, namely the ones that do not have disjoint co-domains. However, $\mathsf{F_i}$ accepts other kinds of intersections which are not possible to express in their calculus. For example merges with type $(\texttt{Int} \to \texttt{Bool})\&(\texttt{Int} \to \texttt{Int})$ are accepted in $\mathsf{F_i}$. Similarly to Castagna et al. previous work [15], their work is focused on combining intersections with functions (but without a merge operator), whereas $\mathsf{F_i}$ is concerned with merges between arbitrary types. Nevertheless, both systems need to express negative information about type variables. That is, which types a given variable cannot be instantiated to. In their calculus, difference takes this role (i.e. $\alpha\setminus\texttt{Int}$); whereas in $\mathsf{F_i}$, one can express it using disjoint quantification (i.e. $\forall(\alpha * \texttt{Int}). \ldots$).

Going in the direction of higher kinds, Compagnoni and Pierce [17] added intersection types to System F_ω and used a new calculus, $\mathsf{F}^\omega_\wedge$, to model multiple inheritance. In their system, types include the construct of intersection of types of the same kind K. Davies and Pfenning [20] studied the interactions between intersection types and effects in call-by-value languages. They proposed a "value restriction" for intersection types, similar to value restriction on parametric polymorphism. None of these calculi has a merge operator.

Recently, some form of intersection types has been adopted in object-oriented languages such as Scala [34], TypeScript [4], Flow [3], Ceylon [1], and Grace [9]. There is also a foundational calculus for Scala that incorporates intersection types [46]. The most significant difference between $\mathsf{F_i}$ and those languages/calculi is that they have no explicit introduction construct like our merge operator. The lack of a native merge operator leads to several ad-hoc solutions for defining a merge operator in those languages, as discussed in Sections 1 and 3.1.

Extensible records. The encoding of multi-field records using intersection types and the merge operator first appeared in Forsythe [45]. Castagna et al. [15] propose an alternative encoding for records. Similarly to $\mathsf{F_i}$'s treatment of elaborating records is Cardelli's work [12] on translating a calculus with extensible records ($\mathsf{F}_{<:\rho}$) to a simpler calculus without records primitives ($\mathsf{F}_{<:}$). However, he does not encode multi-field records as intersections/merges hence his translation is more heavyweight. Crary [19] used intersection types and existential types to address the problem arising from interpreting method dispatch as self-application, but he also did not use intersection types to encode multi-field records.

Wand [49] started the work on extensible records and proposed row types [50] for records, together with a *concatenation* operator, which is used in many calculi with extensible records [27,41,50,48,32,39]. Cardelli and Mitchell [14] defined three primitive operations on records that are also standard in type-systems with record types: *selection*, *restriction*, and *extension*. Several calculi are based on these three primitive operators (especially extension) [42,26,29,30,31,10]. The merge operator in $\mathsf{F_i}$ generalizes the concatenation operator for records, as its components may contain any types (and not just records). Systems with concatenation typically use a set of constraints/filters (such as `lacks` and `contains`) which are useful to combine several, possibly polymorphic, records [31]. In $\mathsf{F_i}$, constraints on labels are encoded using disjoint quantification and intersections. Although systems with records can model structurally typed OO languages, it is harder to encode nominal objects. One advantage of the generality of intersections and merges is that it is easier to have nominal objects. Unlike systems with records, which have been extensively studied, there is much less work on intersection type systems with a merge operator. An interesting avenue for future work is to see whether some known compilation and type-inference techniques for extensible records can be adapted to disjoint intersections and merges.

Traits and mixins. Traits [21,24,33] and mixins [11,6,25,23,5,8] have become very popular in object-oriented languages. They enable restricted forms of multiple inheritance. One of the main differences between traits and mixins are the way in which ambiguity of names is dealt with. Traits reject programs which compose classes with conflicting method implementations, whereas mixins assume a resolution strategy, which is usually language dependent. Our example demonstrated in Section 3 suggests that disjointness in $\mathsf{F_i}$ enables a model with a philosophy similar to traits: merging multiple values of overlapping types is forbidden. However while our simple encodings of objects work for very dynamic forms of prototype inheritance, the work on type systems for mixins/traits usually builds on more conventional class-based OO models.

8 Conclusion and Future Work

This paper described $\mathsf{F_i}$: 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 flexible enough to enable practical applications.

For the future, we intend to create a prototype-based statically typed source language based on $\mathsf{F}_{\mathfrak{i}}$. We are also interested in extending our work to systems with union types and a $\bot$ type. Union types are also widely used in languages such as Ceylon or Flow, but preserving coherence in the presence of union types is challenging. The naive addition of $\bot$ seems to be problematic. The proofs for $\mathsf{F}_{\mathfrak{i}}$ rely on the invariant that a type variable α can never be disjoint to another type that contains α. The addition of $\bot$ breaks this invariant, allowing us to derive, for example, $\Gamma \vdash \alpha * \alpha$.

References

1. Ceylon, https://ceylon-lang.org/
2. The Coq Proof Assistant, https://coq.inria.fr/
3. Flow, https://flowtype.org/
4. TypeScript, https://www.typescriptlang.org/
5. Ancona, D., Lagorio, G., Zucca, E.: Jam—designing a Java extension with mixins. ACM Trans. Program. Lang. Syst. 25(5), 641–712 (2003)
6. Ancona, D., Zucca, E.: An algebraic approach to mixins and modularity. In: ALP'96 (1996)
7. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: POPL'08 (2008)
8. Bettini, L., Bono, V., Likavec, S.: A core calculus of higher-order mixins and classes. In: SAC'04 (2004)
9. Black, A.P., Bruce, K.B., Homer, M., Noble, J.: Grace: The absence of (inessential) difficulty. In: Onward! 2012 (2012)
10. Blume, M., Acar, U.A., Chae, W.: Extensible programming with first-class cases. In: ICFP '06 (2006)
11. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA/ECOOP '90 (1990)
12. Cardelli, L.: Theoretical aspects of object-oriented programming. chap. Extensible records in a pure calculus of subtyping. MIT Press (1994)
13. Cardelli, L., Martini, S., Mitchell, J.C., Scedrov, A.: An extension of System F with subtyping. Inf. Comput. 109(1-2), 4–56 (1994)
14. Cardelli, L., Mitchell, J.C.: Operations on records. In: Mathematical Foundations of Programming Semantics (1990)
15. Castagna, G., Ghelli, G., Longo, G.: A calculus for overloaded functions with subtyping. Information and Computation 117(1), 115–135 (1995)
16. 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: POPL '14 (2014)
17. Compagnoni, A.B., Pierce, B.C.: Higher-order intersection types and multiple inheritance. Math. Structures Comput. Sci. 6(5), 469–501 (1996)
18. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional characters of solvable terms. Mathematical Logic Quarterly 27(2–6), 45–58 (1981)
19. Crary, K.: Simple, efficient object encoding using intersection types. Tech. Rep. CMU-CS-99-100, Cornell University (1998)
20. Davies, R., Pfenning, F.: Intersection types and computational effects. In: ICFP'00 (2000)
21. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28(2), 331–388 (2006)
22. Dunfield, J.: Elaborating intersection and union types. In: ICFP'12 (2012)
23. Findler, R.B., Flatt, M.: Modular object-oriented programming with units and mixins. In: ICFP '98 (1998)
24. Fisher, K.: A typed calculus of traits. In: FOOL11 (2004)
25. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: POPL'98 (1998)
26. Gaster, B.R., Jones, M.P.: A polymorphic type system for extensible records and variants. Tech. Rep. NOTTCS-TR-96-3, University of Nottingham (1996)
27. Harper, R., Pierce, B.: A record calculus based on symmetric concatenation. In: POPL'91 (1991)

28. Harper, R.W., Pierce, B.C.: Extensible records without subsumption. Tech. Rep. CMU-C5-90-102 (1990)
29. Jones, M., Jones, S.P.: Lightweight extensible records for Haskell. Tech. Rep. UU-CS-1999-28, Haskell Workshop (1999)
30. Leijen, D.: First-class labels for extensible rows. Tech. Rep. UU-CS-2004-051, Utrecht University (2004)
31. Leijen, D.: Extensible records with scoped labels. In: Trends in Functional Programming (2005)
32. Makholm, H., Wells, J.B.: Type inference, principal typings, and let-polymorphism for first-class mixin modules. In: ICFP'05 (2005)
33. Odersky, M., Zenger, M.: Scalable component abstractions. In: OOPSLA '05 (2005)
34. Odersky, M., et al.: An overview of the Scala programming language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland (2004)
35. Oliveira, B.C.d.S., Shi, Z., Alpuim, J.: Disjoint intersection types. In: ICFP'16 (2016)
36. Oliveira, B.C.d.S., Van Der Storm, T., Loh, A., Cook, W.R.: Feature-oriented programming with object algebras. In: ECOOP'13 (2013)
37. Pierce, B.C.: Programming with intersection types and bounded polymorphism. Ph.D. thesis, Carnegie Mellon University (1991)
38. Pierce, B.C., Turner, D.N.: Simple type-theoretic foundations for object-oriented programming. J. Funct. Program. 4(2), 207–247 (1994)
39. Pottier, F.: A constraint-based presentation and generalization of rows. In: LICS'03 (2003)
40. Pottinger, G.: A type assignment for the strongly normalizable λ-terms. In: To H. B. Curry: essays on combinatory logic, lambda calculus and formalism (1980)
41. Rémy, D.: Typing record concatenation for free. In: POPL'92 (1992)
42. Rémy, D.: Theoretical aspects of object-oriented programming. chap. Type inference for records in natural extension of ML. MIT Press (1994)
43. Rendel, T., Brachthäuser, J.I., Ostermann, K.: From object algebras to attribute grammars. In: OOPSLA'14 (2014)
44. Reynolds, J.C.: The coherence of languages with intersection types. In: TACS'91 (1991)
45. Reynolds, J.C.: Algol-like languages, chap. Design of the programming language Forsythe. Birkhäuser Boston (1997)
46. Rompf, T., Amin, N.: Type soundness for dependent object types. In: OOPSLA'16 (2016)
47. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: ECOOP'03 (2003)
48. Sulzmann, M.: Designing record systems. Tech. Rep. YALEU/DCS/RR-1128, Yale University (1997)
49. Wand, M.: Complete type inference for simple objects. In: LICS'87 (1987)
50. Wand, M.: Type inference for record concatenation and multiple inheritance. In: LICS'89 (1989)