Disjoint Intersection Types and Disjoint Quantification

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

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 [14] 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* [32] in various communities. Currently there are various solutions in functional programming languages [30,27], object-oriented programming languages [31,34,26,18] and theorem provers [12,28]. 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 [19]. 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 [19] 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 [8]. 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 [14] 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 [14] 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 [24] (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 $\mathsf{F}_{\mathsf{i}}^{*}$. $\mathsf{F}_{\mathsf{i}}^{*}$ 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.

$\mathrm{F}^{*}_{\mathrm{i}}$'s type system only accepts programs that use disjoint intersection types. As shown in Section 5 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 [4], 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: Extensibility

Various solutions to the Expression Problem [32] in the literature [29,5,26,18,12] are closely related to type-theoretic encodings of datatypes. Indeed, variants of

the same idea keep appearing in different programming languages, because the encoding of the idea needs to exploit the particular features of the programming language (or theorem prover). Unfortunately language-specific constructs obscure the key ideas behind those solutions.

In this section we show a solution to the Expression Problem that intends to capture the key ideas of various solutions in the literature. Moreover, it is shown how *all the features* of $\mathsf{F}_{\mathsf{i}}^{*}$ (intersection types, the merge operator, parametric polymorphism and disjoint quantification) are needed to properly encode one important combinator [19] used to compose multiple operations over datatypes.

3.1 Church Encoded Arithmetic Expressions

In the Expression Problem, the idea is to start with a very simple system modeling arithmetic expressions and evaluation. The standard typed Church encoding [3] for arithmetic expressions, denoted as the type `CExp`, is:

```
type CExp = ∀E. (Int → E) → (E → E → E) → E
```

However, as done in various solutions to extensibility, it is better to break down the type of the Church encoding into two parts:

```
type ExpAlg[E] = {
  lit: Int → E,
  add: E → E → E
} in ...
```

The first part, captured by the type `ExpAlg[E]` is constitutes the so-called algebra of the datatype. For additional clarity of presentation, records (supported in the implementation of $\mathsf{F}_{\mathsf{i}}^{*}$) are used to capture the two components of the algebra. The first component abstracts over the type of the constructor for literal expressions (`Int` → `E`). The second component abstracts over the type of addition expressions (`E` → `E` → `E`).

The second part, which is the actual type of the Church encoding, is:

```
type Exp = { accept: ∀E. ExpAlg[E] → E } in ...
```

It should be clear that, modulo some refactoring, and the use of records, the type `Exp` and `CExp` are equivalent.

Data constructors. Using `Exp` the two data constructors are defined as follows:

```
let lit (n: Int): Exp = {
  accept = ΛE → fun (f: ExpAlg[E]) → f.lit n
} in
let add (e1: Exp) (e2: Exp): Exp = {
  accept = ΛE → fun (f: ExpAlg[E]) →
    f.add (e1.accept[E] f) (e2.accept[E] f)
} in
...
```

Note that the notation `ΛE` in the definition of the `accept` fields is a type abstraction: it introduces a type variable in the environment. The definition of the constructors themselves follows the usual Church encodings.

Simple expressions, can be built using the data constructors:

```
let five : Exp = add (lit 3) (lit 2) in ...
```

Operations. Defining operations over expressions requires implementing `ExpAlg[E]`. For example, an interesting operation over expressions is evaluation. The first step is to define the evaluation operation is to chose how to instantiate the type parameter `E` in `ExpAlg[E]` with a suitable concrete type for evaluation. One such suitable type is:

```
type IEval = { eval: Int } in ...
```

Using `IEval`, a record `evalAlg` implementing `ExpAlg` is defined as follows:

```
let evalAlg: ExpAlg[IEval] = {
  lit = fun (x: Int) → { eval = x },
  add = fun (x: IEval) (y: IEval) → {
    eval = x.eval + y.eval
  }
} in ...
```

In this record, the two operations `lit` and `add` return a record with type `IEval`. The definition of `eval` for `lit` and `add` is straightforward.

Using `evalAlg`, the expression `five` can be evaluated as follows:

```
(five.accept[IEval] evalAlg).eval
```

3.2 Extensibility and Subtyping

Of course, in the Expression Problem the goal is to achieve extensibility in two dimensions: constructors and operations. Moreover, in the presence of subtyping it is also interesting to see how the extended datatypes relate to the original datatypes. We discuss the two topics next.

New constructors. Here is the code needed to add a new subtraction constructor:

```
type SubExpAlg[E] = ExpAlg[E] & { sub: E → E → E } in
type SubExp = { accept: ∀A. SubExpAlg[A] → A } in
let sub (e1: SubExp) (e2: SubExp): SubExp = {
  accept = ΛE → fun (f : SubExpAlg[E]) →
    f.sub (e1.accept[E] f) (e2.accept[E] f)
} in
...
```

Firstly `SubExpAlg` defines an extended algebra that contains the constructors of `ExpAlg` plus the new subtraction constructor. Intersection types are used to do the type composition. Secondly, a new type of expressions with subtraction (`SubExp`) is needed. For `SubExp` it is important that the `accept` field now takes an algebra of type `SubExpAlg` as argument. This is necessary to define the constructor for subtraction (`sub`), which requires the algebra to have the field `sub`.

Extending existing operations. In order to use evaluation with the new type of expressions, it is necessary to also extend evaluation. Importantly, extension is achieved using the merge operator:

```
let subEvalAlg = evalAlg ,, {
  sub = fun (x: IEval) (y: IEval) → {
    eval = x.eval - y.eval
  }
} in ...
```

In the code, the merge operator takes `evalAlg` and a new record with the implementation of evaluation for subtraction, to define the implementation for arithmetic expressions with subtraction.

Subtyping. In the presence of subtyping, there are interesting subtyping relations between datatypes and their extensions [26]. Such subtyping relations are usually not discussed in theoretical treatments of Church encodings. This is probably partly due to most work on typed Church encodings being done in calculi without subtyping.

The interesting aspect about subtyping in typed Church encodings is that subtyping follows the opposite direction of the extension. In other words subtyping is contravariant with respect to the extension. Such contravariance is explained by the type of the `accept` field, which is a function where the argument type is refined in the extensions. Thus, due to the contravariance of subtyping on functions, the extension becomes a supertype of the original datatype.

In the particular case of expressions `Exp` (the original and smaller datatype) is a subtype of `SubExp` (the larger and extended datatype). Because of this subtyping relation, writing the following expression is valid in $\mathsf{F}^{*}_{\mathsf{i}}$:

```
let three : SubExp = sub five (lit 2)
```

Note the `three` is of type `SubExp`, but the first argument (`five`) to the constructor `sub` is of type `Exp`. This can only type-check if `Exp` is indeed a subtype of `SubExp`.

New operations. The second type of extension is adding a new operation, such as pretty printing. Similarly to evaluation, the interface of the pretty printing feature is modeled as:

```
type IPrint = { print: String } in ...
```

The implementation of pretty printing for expressions that support literals, addition, and subtraction is:

```
let printAlg: SubExpAlg[IPrint] = {
  lit = fun (x: Int) → {
    print = x.toString()
  },
  add = fun (x: IPrint) (y: IPrint) → {
    print = x.print ++ " + " ++ y.print
  },
  sub = fun (x: IPrint) (y: IPrint) → {
    print = x.print ++ " - " ++ y.print
```

```
  }
} in ...
```

The definition of `printAlg` is unremarkable. With `printAlg` we can pretty print the expression represented by `three`:

```
(three.accept[IPrint] printAlg).print
```

3.3 Composition of Algebras

The final example shows a non-trivial combinator for algebras that allows multiple algebras to be combined into one. A version of this combinator has been encoded in Scala before using intersection types (which Scala supports) and an encoding of the merge operator [19,23]. Unfortunatelly, the Scala encoding of the merge operator is quite complex as it relies on low-level type-unsafe programming features such as dynamic proxies, reflection or other meta-programming techniques. In $\mathsf{F}_{\mathsf{i}}^{*}$ there is no need for such hacky encoding, as the merge operator is natively supported. Therefore the combinator for composing algebras is implemented much more elegantly. The combinator is defined by the `combine` function, which takes two object algebras to create a combined algebra. It does so by constructing a new algebra where each field is a function that delegates the input to the two algebra parameters.

```
let combine A (B * A) (f: ExpAlg[A]) (g: ExpAlg[B]) :
  ExpAlg[A & B] = {
    lit = fun (x: Int) → f.lit x ,, g.lit x,
    add = fun (x: A & B) (y: A & B) →
      f.add x y ,, g.add x y
}
```

Note how `combine` requires all the interesting features of $\mathsf{F}_{\mathsf{i}}^{*}$. Parametric polymorphism is needed because `combine` must compose algebras with arbitrary type parameters. Intersection types are needed because the resulting algebra will create values with an intersection type composing the two type parameters of the two input algebras. The merge operator is needed to compose the results of each algebra together. Finally, a disjointness constraint is needed to ensure that the two input algebras build values of disjoint types (otherwise ambiguity could arize).

With `combine` printing and evaluation of expressions with subtraction is done as follows:

```
let newAlg: ExpAlg[IEval&IPrint] =
    combine[IEval,IPrint] evalAlg printAlg in
let o = five.accept[IEval&IPrint] newAlg in
o.print ++ " = " ++ o.eval.toString()
```

Note that `o` is a value that supports both evaluation and printing. The final expression uses `o` for doing both printing and evaluation.

4 The F_i Calculus

This section presents the syntax, subtyping, and typing of F_i: a calculus with intersection types, parametric polymorphism and a merge operator. This calculus is an extension of λ_i, which in its turn is heavily based on Dunfield's calculus [14]. We will show that F_i is type-safe and it preserves the coherent semantics of λ_i. Section 5 introduces the necessary changes to the disjointness concept in order to retain coherence.

All the meta-theory has been mechanized in Coq, which we refer to the supplementary materials submitted with the paper.

4.1 Syntax

Figure 1 shows the syntax of F_i. The differences to λ_i are highlighted in gray.

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

Fig. 1. F_i syntax.

Types. Metavariables A, B range over types. Types include all constructs in λ_i: a top type $\top$; the type of integers $\texttt{Int}$; function types $A \to B$; and intersection types $A\&B$. This calculus is extended with two standard constructs in System F: type variables α and type abstraction $\forall\alpha.\, A$. The latter includes an extra disjointness constraint tied to the type variable α.

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 $\lambda(x{:}A).\, e$; 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 $e1 \,\mathbin{\mathbb{M}}\, e2$, corresponding to intersections of types A&B. This calculus is extended with two standard constructs in System F: abstraction of type variables over terms $\Lambda\alpha.\, e$; and and application of terms to types $e\ A$. The former includes an extra disjointness constraint tied to the type variable α.

Contexts. Typing contexts Γ track bound type variables α with disjointness cosntraint 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 (Sα) 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 hypothesis are necessary to ensure coherence, as explained in .... We will come back to this in the next section. Finally, in (S$\forall$) a universal quantifier ($\forall$) is covariant in its body, and contravariant in its disjointness constraints. As with the subtyping relation defined in ..., we can show that subtyping defined by $<:$ is also reflexive and transitive.

4.3 Typing

The well-formedness rules are shown in the top part of Figure 3. In addition to the original rules, (WF$\forall$) is not surprising. Note how we ensure the well-formedness of the constraint. 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

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

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

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

$$\frac{}{A <: \top \;\hookrightarrow \lambda(x{:}|A|).\,()}\;\mathrm{S}\top \qquad \frac{}{\mathtt{Int} <: \mathtt{Int} \;\hookrightarrow \lambda(x{:}|\alpha|).\,x}\;\mathrm{S}\mathbb{Z}$$

$$\frac{}{\alpha <: \alpha \;\hookrightarrow \lambda(x{:}|\alpha|).\,x}\;\mathrm{S}\alpha$$

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

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

$$\frac{A_1 <: A_3 \;\hookrightarrow E \qquad A_3 \text{ ordinary}}{A_1 \& A_2 <: A_3 \;\hookrightarrow \lambda(x{:}|A_1 \& A_2|).\,[\![A_3]\!]_{(E\ (\mathtt{proj}_1\ x))}}\;\mathrm{S}\&\mathrm{L}_1$$

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

$$\frac{B_1 <: B_2 \;\hookrightarrow E \qquad A_2 <: A_1 \;\hookrightarrow E_d}{\forall(\alpha * A_1).\,B_1 <: \forall(\alpha * A_2).\,B_2 \;\hookrightarrow \lambda(f{:}|\forall\alpha.\,B_1|).\,\Lambda\alpha.\,E\ (f\ \alpha)}\;\mathrm{S}\forall$$

Fig. 2. Subtyping rules of F_{i}.

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

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

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

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

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

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x \Rightarrow A \hookrightarrow x}\ \text{T-Var}$$

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

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

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

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

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

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

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

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

Fig. 3. Type system of F_i.

rule (T-Sub). The new rules, inspired on the standard System-F, are the check rule for type application (T-TApp), and the infer rule 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.

5 Semantics, Disjointness and Coherence

5.1 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 [2].

Target language. The syntax and typing of our target language is unsurprising. The syntax of the target language is shown in Figure 4. The highlighted part shows its difference with the standard System F. The typing rules can be found in the appendix.

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

Fig. 4. Target language syntax.

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

Type and context translation. Figure 5 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.

$\boxed{|A| = T}$

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

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

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

Fig. 5. Type and context translation.

5.2 Top-like types and their coercions

TODO

Coercions TODO

5.3 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 1). For example,

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

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

$\boxed{\rceil A \lceil}$

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

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

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

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

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

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

Fig. 6. Top-like types and respective coercions.

In rules (Sα), (S$\forall$), coercions are just identity functions. In (S$\to$), we elaborate the subtyping of parameter and return types by η-expanding f to $\lambda(x : |A_3|).\, f\ x$, 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. Note that the two rules are overlapping and hence a program can be elaborated differently, depending on which rule is used. Finally, all rules produce type-correct coercions:

Lemma 1 (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[3].

Ordinary types Atomic types are just those which are not intersection types, and are asserted by the judgement

$$A \text{ ordinary}$$

In the left decomposition rules for intersections we introduce a requirement that A_3 is atomic. The consequence of this requirement is that when A_3 is an

[3] The proofs of major lemmata and theorems can be found in the appendix.

intersection type, then the only rule that can be applied is (S&R). With the atomic constraint, one can guarantee that at any moment during the derivation of a subtyping relation, at most one rule can be used.

Unique coercions TODO More formally, as also shown in ..., with disjoint intersections the following theorem holds:

Lemma 2 (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 3 (Unique coercion).
If $A <: B \hookrightarrow E_1$ *and* $A <: B \hookrightarrow E_2$*, where* A *and* B *are well-formed types, then* $E_1 \equiv E_2$.

Expressiveness. Remarkably, our restrictions on subtyping do not sacrifice the expressiveness of subtyping since we have the following two theorems:

Theorem 1. *If* $A_1 <: A_3$*, then* $A_1 \& A_2 <: A_3$.

Theorem 2. *If* $A_2 <: A_3$*, then* $A_1 \& A_2 <: A_3$.

The interpretation of the theorem is that: even though the premise is made more strict by the atomic condition, we can still derive the every subtyping relation in the unrestricted system.

5.4 Disjointness

Throughout the paper we already presented an intuitive definition for disjoiness. Here such definition is made a bit more precise, and well-suited to F^*_i.

Definition 2 (Disjoint types).
Given a context Γ*, two types* A *and* B *are said to be disjoint (written* $\Gamma \vdash A * B$*) if they do not share a common supertype. That is, there does not exist a type* C *such that* $A <: C$ *and that* $B <: C$*. Note that we assume that all free type variables in* A*,* B *and* C *are bound in* Γ.

$$\Gamma \vdash A * B \equiv \nexists C.\ A <: C \wedge B <: C$$

To see this definition in action, `Int` and `Char` are disjoint, because there is no type that is a supertype of the both. On the other hand, `Int` is not disjoint with itself, because `Int` <: `Int`. This implies that disjointness is not reflexive as subtyping is. Two types with different shapes are always disjoint, unless one of them is an intersection type. For example, a function type and a universally quantified type must be disjoint. But a function type and an intersection type may not be. Consider:

$$\texttt{Int} \to \texttt{Int} \quad \text{and} \quad (\texttt{Int} \to \texttt{Int}) \& (\texttt{String} \to \texttt{String})$$

Those two types are not disjoint since $\texttt{Int} \to \texttt{Int}$ is their common supertype.

5.5 Well-formedness and substitution

TODO

5.6 Bi-directional type-system with Elaboration

TODO: talk about bi-directional type-system.

Key idea of the translation. This translation turns merges into usual pairs, similar to Dunfield's elaboration approach [14]. For example,

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

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

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

This expression will be translated to

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

The coercion in this case is $(\lambda(x : (\texttt{Int}, \texttt{String})).\, \texttt{proj}_2 x)$. 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 $|A|$. In $\mathsf{F}_\mathtt{i}$, 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. (T-Merge) straighforwardly translates merges into pairs.

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

Theorem 3 (Type preservation).
If $\Gamma \vdash e : A \hookrightarrow E$, *then* $|\Gamma| \vdash E : |A|$.

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

Theorem 4 (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.

Although the system shown in the Section 4 is type-safe, it is not coherent. This section shows how to modify F_i so that it guarantees coherence as well as type soundness. The result is a calculus named F_i^*. The keys aspects are the notion of disjoint intersections, and disjoint quantification for polymorphic types.

Well-formedness. We require that the two types of an intersection must be disjoint in their context, and that the disjointness constraint in a universal type is well-formed. Under the new rules, intersection types such as `Int&Int` are no longer well-formed because the two types are not disjoint.

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 makes sure that the type argument satisfies the disjointness constraint.

Metatheory. Since in this section we only restrict the type system in the previous section, it is easy to see that type preservation and type-safety still holds. Additionally, we can show that typing always produces a well-formed type by proving the following results.

Lemma 4 (Instantiation).
If $\Gamma, \alpha * B \vdash C$, $\Gamma \vdash A$, $\Gamma \vdash A * B$ *then* $\Gamma \vdash [\alpha := A]\ C$.

Lemma 5 (Well-formed typing).
If $\Gamma \vdash e : A$, *then* $\Gamma \vdash A$.

Proof. By induction on the derivation that leads to $\Gamma \vdash e : A$ and applying Lemma 4 in the case of (T-TAPP).

With our new definition of well-formed types, this result is nontrivial. In general, disjointness judgements are not invariant with respect to free-variable substitution. In other words, a careless substitution can violate the disjoint constraint in the context. For example, in the context $\alpha * \texttt{Int}$, α and `Int` are disjoint:

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

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

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

is longer true since `Int` is clearly not disjoint with itself.

5.7 Coherence of the Elaboration

Combining the previous results, we are able to show the central theorem:

Theorem 5 (Unique elaboration).
If $\Gamma \vdash e : A_1 \hookrightarrow E_1$ *and* $\Gamma \vdash e : A_2 \hookrightarrow E_2$, *then* $E_1 \equiv E_2$. *("$\equiv$" means syntactical equality, up to α-equality.)*

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 5, 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 3 and conclude that E is unique.

6 Disjointness

Section 4 presented a type system with disjoint intersection types and disjoint quantification that is both type-safe and coherent.

This section 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. Therefore we solve the problem of finding an algorithm to compute disjointness.

6.1 Remarks on Top

TODO should this go in the overview?

6.2 Dropping the specification

TODO should this go in the overview?

6.3 Algorithmic Rules

The rules for the disjointness judgement are shown in Figure 7, which consists of two judgements.

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

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

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

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

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

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

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

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

Fig. 7. Algorithmic Disjointness.

Main judgement. The judgement $\Gamma \vdash A *_i B$ 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 $(*\top\mathsf{Sym})$ say that any type is disjoint to $\top$. $(*\alpha)$ is the base rule and $(*\alpha\mathsf{Sym})$ is its twin rule. Both rules state that a type variable is disjoint to some type A, if Γ contains any subtype of the corresponding disjointness constraint. This rule is a specialization of the more general lemma:

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

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 both constraints into Γ and checks for disjointness in the bodies under that environment. The remaining rules are identical to the original rules, and we will only briefly explain them. The rule for functions $((*\to))$ says that two function types are disjoint if and only if their return types are disjoint. Finally 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. Up till now, the rules of $\Gamma \vdash A *_{\mathrm{i}} B$ have only taken care of two types with the same language constructs. But how can be the fact that `Int` and `Int` $\rightarrow$ `Int` are disjoint be decided? That is exactly the place where the judgement $A *_{\mathrm{ax}} B$ comes in handy. It provides the axioms for disjointness. What is captured by the set of rules is that $A *_{\mathrm{ax}} B$ holds for all two types of different constructs unless any of them is an intersection type. That is because for example, `Int`&(`Char` $\rightarrow$ `Char`) and `Char` $\rightarrow$ `Char` use different constructs and yet are not disjoint. There are two points worth noting. One is that the only type that is disjoint with itself is $\bot$: the type which has no values. The other point is that all rules need a dual form to ensure symmetry. The rule ($*$AxSym) takes care of that.

6.4 Metatheory

The algorithmic rules for disjointness are sound and complete.

Theorem 6 (Soundness of algorithmic disjointness).
For any two types A *and* B, $\Gamma \vdash A *_{\mathrm{i}} B$ *implies* $\Gamma \vdash A * B$.

Proof. By induction on the derivation of $\Gamma \vdash A *_{\mathrm{i}} B$.

Theorem 7 (Completeness of algorithmic disjointness).
For any two types A, B, $\Gamma \vdash A * B$ *implies* $\Gamma \vdash A *_{\mathrm{i}} B$.

Proof. By a case analysis on the shape of A and B.

7 Related Work

Coherence Reynolds invented Forsythe [25] in the 1980s. Our merge operator is analogous to his operator p_1, p_2. Forsythe has a coherent semantics. The result was proved formally by Reynolds [24] 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 $\mathsf{F}^*_{\mathrm{i}}$ 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 [20] made a comprehensive review of coherence, especially on Curien and Ghelli [9] 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 [14]. 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 [21] 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 [20] and a 1997 report [22].

Going in the direction of higher kinds, Compagnoni and Pierce [7] added intersection types to System F_{ω} and used the new calculus, $\mathsf{F}_{\wedge}^{\omega}$, to model multiple inheritance. In their system, types include the construct of intersection of types of the same kind K. Davies and Pfenning [11] 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. [6] 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 3, this feature is pivotal in supporting modularity and extensibility because it allows dynamic composition of values.

Other type systems with intersection types. Refinement intersection [13,10,15] 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. [16] 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 [32]. 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 [33] can be used to implement type-theoretic encodings of datatypes [17]. It has been shown [5] 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 [26,18]. It is even possible to solve the Expression Problem in theorem provers like Coq, by exploiting Coq's type class mechanism [12]. 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 3, 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.

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., Martini, S., Mitchell, J.C., Scedrov, A.: An extension of system f with subtyping. Inf. Comput. 109(1-2) (Feb 1994)
5. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. 19(5) (2009)
6. 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)
7. Compagnoni, A.B., Pierce, B.C.: Higher-order intersection types and multiple inheritance. Mathematical Structures in Computer Science (1996)
8. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional characters of solvable terms. Mathematical Logic Quarterly (1981)
9. 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)
10. Davies, R.: Practical refinement-type checking. Ph.D. thesis, University of Western Australia (2005)
11. Davies, R., Pfenning, F.: Intersection types and computational effects. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00) (2000)
12. 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)
13. Dunfield, J.: Refined typechecking with stardust. In: Proceedings of the 2007 workshop on Programming languages meets program verification. ACM (2007)
14. Dunfield, J.: Elaborating intersection and union types. Journal of Functional Programming (2014)
15. 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)
16. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM (JACM) (2008)
17. Hinze, R.: Generics for the masses. J. Funct. Program. 16(4-5) (Jul 2006)
18. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses. In: ECOOP 2012–Object-Oriented Programming (2012)
19. 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)
20. Pierce, B.C.: Programming with intersection types and bounded polymorphism. Ph.D. thesis, Carnegie Mellon University Pittsburgh, PA (1991)

21. Pierce, B.C.: Programming with intersection types, union types, and polymorphism (1991)
22. Pierce, B.C.: Intersection types and bounded polymorphism. Mathematical Structures in Computer Science (1997)
23. Rendel, T., Brachthäuser, J.I., Ostermann, K.: From object algebras to attribute grammars. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '14 (2014)
24. 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)
25. Reynolds, J.C.: Design of the programming language Forsythe (1997)
26. 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)
27. 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)
28. 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)
29. Swierstra, W.: Data types à la carte. J. Funct. Program. 18(4) (Jul 2008)
30. Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423–436 (July 2008)
31. 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)
32. Wadler, P.: The expression problem. Java-genericity mailing list (1998)
33. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: POPL. pp. 60–76. ACM (1989)
34. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: FOOL (2005)