Disjoint Intersection Types and Disjoint Quantification

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

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 $F_{\&}$: a core calculus with a variant of *intersection types*, *parametric polymorphism* and a *merge operator*. The semantics $F_{\&}$ 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.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms Design, Languages, Theory

Keywords Intersecion Types, Polymorphism, Type System

1. Introduction

Previous work by Dunfield [15] has shown the power of intersection types and a merge operator. The presence of a merge operator in a core calculus provides significant expressiveness, allowing encodings for many other language constructs as syntactic sugar. For example single-field records are easily encoded as types with a label, and multi-field records are encoded as the concatenation of single-field records. Concatenation of records is expressed using intersection types at the type-level and the corresponding merge operator at the term level. Dunfield formalized a simply typed lambda calculus with intersection types and a merge operator. He showed how to give a semantics to the calculus by a type-directed translation to a simply typed lambda calculus extended with pairs. The type-directed translation is simple, elegant, and type-safe.

Intersection types and the merge operator are also useful in the context of software *extensibility*. In recent years there has been a wide interest in presenting solutions to the *expression problem* [36] in various communities. Currently there are various solutions in functional programming languages [5, 34], object-oriented programming languages [10, 22, 35, 37] and theorem provers [13, 31]. Many of the proposed solutions for extensibility are closely related to type-theoretic encodings of datatypes [4], 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 [23]. 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 polymorhism. 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 disavantage 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 arizes 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 $F_{\&}$: a core calculus with a variant of *intersection types*, *parametric polymorphism* and a *merge operator*. The semantics $F_{\&}$ is both type-safe and coherent. Thus $F_{\&}$ 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 ($A * B$) if there is no type C such that both A and B are subtypes of C. Formally this definition is captured as follows:

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

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.

We also show that disjoint intersection types can be extended to support 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.

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 $F_{\&}$ and Proof of Coherence:** An elaboration semantics of System $F_{\&}$ into System F is given. Type-soundness and coherence are proved.
- **Extensible Type-Theoretic Encodings:** We show that in $F_{\&}$ type-theoretic encodings can be combined with subtyping to provide extensibility.
- **Implementation:** An implementation of an extension of System $F_{\&}$, as well as the examples presented in the paper, are publicly available[1].

2. Overview

This section introduces $F_{\&}$ 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. Finally we illustrate the expressive power of $F_{\&}$ by encoding extensible type-theoretic encodings of datatypes.

Note that this section uses some syntactic sugar, as well as standard programming language features, to illustrate the various concepts in $F_{\&}$. Although the minimal core language that we formalize in Section 4 does not present all such features, our implementation supports them.

BRUNO: Need to type-check the programs! BRUNO: Need to make code formatting homogeneous!

2.1 Intersection Types and the Merge Operator

Intersection types date back as early as Coppo et al.'s work [9]. 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 $F_{\&}$) contains exactly those values which can be used as either of type A or of type B. For instance, consider the following program in $F_{\&}$:

```
let x : Int & Char = ...;   -- definition ommitted
let idInt (y : Int) : Int = y;
let idChar (y : Char) : Char = y;
(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 deliberatly 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 [15] 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 $F_{\&}$ as follows:

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

In $F_{\&}$ (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');
```

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:

$$(\lambda(x : \mathsf{Int}).x)\ (1 ,, 2)$$

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 [15] 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 take either always take the left-most value matching the type in the merge. Similarly, one could take always take the right-most value matching the type in the merge. Either way, the meaning of a program will always depend on a biased implementation choice, which is clearly unsatisfying from the theoretical point of view (although perhaps acceptable in practice). Moreover, even if it 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.

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

2.3 Restoring Coherence: Disjoint Intersection Types

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

This is precisely the approach taken in $F_{\&}$. $F_{\&}$ 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 $A * B$, we would have:

$$A * B \equiv A \not<: B \wedge B \not<: A$$

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:

$$((1 \;,,\; 'c') \;,,\; (2 \;,,\; \mathsf{True}))$$

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

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

Nevertheless the following program still leads to incoherence:

$$(\lambda(x{:}\mathsf{Int}).\,x)\ ((1,,'c'),,(2,,\mathsf{True}))$$

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 $A * B$) if there is no type C such that both A and B are subtypes of C:

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

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

$F_{\&}$'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 guaranteing that the semantics is coherent.

2.4 Parametric Polymorphism and Intersection Types

Before we show how $F_{\&}$ extends the idea of disjointness to parametric polymorphism, we discuss some non-trivial issues that arize from the interaction between parametric polymorphism and intersection types. Consider the attempt to write the following polymorphic function in $F_{\&}$:

$$\textbf{let}\ \mathtt{fst}\ A\ B\ (x : A\&B) = (\lambda z : A.z)\ x;$$

The $\mathtt{fst}$ function is supposed to extract a value from the merge that matches the type of the first type parameter (A). However this function is problematic. The reason is that when A and B are instantiated to non-disjoint types, then uses of $\mathtt{fst}$ may lead to incoherence. For example, consider the following use of $\mathtt{fst}$:

$$\mathtt{fst}\ \mathtt{Int}\ \mathtt{Int}\ (1,,2)$$

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 $\mathtt{fst}$.

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). Unfortunatelly, 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 $\mathtt{fst}$ above is 2. Indeed simple equational reasoning seems to validate such result:

$\mathtt{fst}\ \mathtt{Int}\ \mathtt{Int}\ (1,,2)$
$\equiv$ {definition of $\mathtt{fst}$}
$(\lambda(z : \mathtt{Int}).z)(1,,2)$
$\equiv$ {implicit coercion (biased to the right)}
$(\lambda(z : \mathtt{Int}).z)\ 2$
$\equiv$ {β-reduction}
2

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 $\mathtt{fst}$, lookup is triggered by a coercion function inserted in the definition of $\mathtt{fst}$ at compile-time. In the definition of $\mathtt{fst}$ 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. Unfortunatelly, 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!

Unfortunatelly 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 $F_{\&}$.

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 $\mathtt{fst}$ would be rejected, since there are indeed some cases that may lead to incoherent programs. Unfortunatelly this is too restrictive and prevents many useful programs using both parametric polymorphism and intersection types. In particular, in the case of $\mathtt{fst}$, 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, $F_{\&}$ uses an extension to universal quantification called *disjoint quantification*. Inspired by bounded quantification [6], 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 $F_{\&}$, would be written as:

$$\textbf{let}\ \mathtt{fst}\ A\ (\ B{*}A\ )\ (x : A\&B) = (\lambda z : A.z)\ x;$$

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`:

$$\mathtt{fst}\ \mathtt{Int}\ \mathtt{Char}\ (1,,'c')$$

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:

$$\mathtt{fst}\ \mathtt{Int}\ \mathtt{Int}\ (1,,2)$$

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 [36] in the literature [7, 10, 13, 22, 33] 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). Unfortunatelly 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 $F_{\&}$ (intersection types, the merge operator, parametric polymorphism and disjoint quantification) are needed to properly encode one important combinator [23] 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 [4] for arithmetic expressions, denoted as the type `CExp`, is:

```
type CExp = forall 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
};
```

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 $F_{\&}$) 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 : forall E. ExpAlg[E] -> E};
```

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 = /\E -> \(f: ExpAlg[E]) -> f.lit n
};
let add (e1: Exp) (e2: Exp): Exp = {
  accept = /\E -> \(f: ExpAlg[E]) ->
    f.add (e1.accept[E] f) (e2.accept[E] f)
};
```

Note that the notation `/\E` 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)
```

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};
```

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

```
let evalAlg: ExpAlg[IEval] = {
  lit = \(x: Int) -> {eval = x},
  add = \(x: IEval) (y: IEval) -> {
    eval = x.eval + y.eval
  }
};
```

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};

type SubExp = {
  accept: forall A. SubExpAlg[A] -> A
};

let sub (e1: SubExp) (e2: SubExp): SubExp ={
  accept = /\E -> \(f : SubExpAlg[E]) ->
  f.sub (e1.accept[E] f) (e2.accept[E] f)
};
```

Firstly SubExpAlg defines an extended algebra that contains the constructors of ExpAlg plus the new subtraction constructors. 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 = \(x: IEval) (y: IEval) -> {
    eval = x.eval - y.eval
  }
};
```

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 [10]. 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, writting the following expression is valid in $F_{\&}$:

```
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};
```

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

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

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 [23, 28]. 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 metaprogramming techniques. In $F_{\&}$ 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 = \(x: Int) -> f.lit x ,, g.lit x,
    add = \(x: A & B) (y: A & B) ->
      f.add x y ,, g.add x y
}
```

Note how combine requires all the interesting features of $F_{\&}$. 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;
let o = five.accept[IEval&IPrint] newAlg;
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 $F_{\&}$ Calculus

This section presents the syntax, subtyping, typing, as well as the (incoherent) semantics of $F_{\&}$: a calculus with intersection types, parametric polymorphism and a merge operator. Section 5 shows the necessary changes to this calculus for supporting disjoint intersection types and disjoint quantificantion. BRUNO: I liked the name that you gave this intermediate calculus before. It is confusing to have the same name for both calculus.

GEORGE: Read TAPL chapter on bounded quantification.

GEORGE: Add context to the subtyping rules.

4.1 Syntax

Figure 1 shows the syntax of $F_{\&}$. The differences with System F, highlighted in gray, are intersection types $A\&B$ at the type-level and the "merges" $e_1 ,, e_2$ at the term level.

Types Metavariables A, B range over types. Types include standard constructs in System F: type variables α; function types $A \to B$; and type abstraction $\forall\alpha.A$. $A\&B$ denotes the intersection of types A and B.

Terms Metavariables e range over terms. Terms include standard constructs in System F: variables x; abstraction of terms over variables of a given type $\lambda(x:A).e$; application of terms e_1 to terms e_2, written $e_1\ e_2$; abstraction of type variables over terms $\Lambda\alpha.e$;

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

Figure 1. $F_{\&}$ syntax.

and application of terms to types $e\ A$. The expression $e_1 ,, e_2$ is the *merge* of two terms e_1 and e_2. Merges of terms correspond to intersections of types $A\&B$.

Contexts Typing contexts Γ track bound type variables α 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 the top part of Figure 2. At the moment, the reader is advised to ignore the gray-shaded part in the rules, which will be explained later. The rule «SUB_FUN» says that a function is contravariant in its parameter type and covariant in its return type. In «SUB_FORALL» a universal quantifier ($\forall$) is covariant in its body. 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 «SUB_INTER» 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.

Metatheory As other sane subtyping relations, we can show that subtyping defined by $<:$ is also reflexive and transitive.

Lemma 1 (Subtyping is reflexive)**.** *For all type* A, $A <: A$.

Lemma 2 (Subtyping is transitive)**.** *If* $A_1 <: A_2$ *and* $A_2 <: A_3$, *then* $A_1 <: A_3$.

For the corresponding mechanized proofs in Coq, we refer to the supplementary materials submitted with the paper.

4.3 Typing

The well-formedness rules are shown in the middle part of Figure 2. In addition to the standard rules, «F_WF_INTER» is also not surprising. 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 judgement is of the form:

$$\Gamma \vdash e : A$$

It reads: "in the typing context Γ, the term e is of type A" The rules that are the same as in System F are rules for variables «F_TY_VAR», lambda abstractions «F_TY_LAM», and type applications «F_TY_TAPP». For the ease of discussion, in «F_TY_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. The rule «F_TY_APP» needs special attention as we add a subtyping requirement: the type of the argument (A_3) is a subtype of the type of the parameter (A_1). «F_TY_MERGE» means that a merge $e_1 ,, e_2$, is assigned an intersection type composed of the resulting types of e_1 and e_2.

4.4 Semantics

We define the dynamic semantics of the call-by-value $F_{\&}$ by means of a type-directed translation to an extension of System F [3].

Target language The syntax and typing of our target language is unsurprising. The syntax of the target language is shown in Figure 3. The highlighted part shows its difference with the standard System F. The typing rules can be found in the appendix.

Key Idea of the Translation This translation turns merges into usual pairs, similar to Dunfield's elaboration approach [15]. For example,

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

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

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

This expression will be translated to

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

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

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

GEORGE: Also need two versions for this.

Coercive subtyping BRUNO: Figure needs to be updated with Γ

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 3). For example,

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

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

In rules «SUB_VAR», «SUB_FORALL», coercions are just identity functions. In «SUB_FUN», 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 «SUB_INTER_1», «SUB_INTER_2», and «SUB_INTER» elaborate intersection types. «SUB_INTER» uses both coercions to form a pair. Rules «SUB_INTER_1» and «SUB_INTER_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.

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

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

$$\frac{}{\Gamma \vdash \alpha <: \alpha \hookrightarrow \lambda(x{:}|\alpha|).\, x}\ \textsc{Sub\_Var} \qquad \frac{\Gamma \vdash B_1 <: A_1 \hookrightarrow E_1 \qquad \Gamma \vdash A_2 <: B_2 \hookrightarrow E_2}{\Gamma \vdash A_1 \to A_2 <: B_1 \to B_2 \hookrightarrow \lambda(f{:}|A_1 \to A_2|).\, \lambda(x{:}|B_1|).\, E_2\ (f\ (E_1\ x))}\ \textsc{Sub\_Fun}$$

$$\frac{A_1 <: [\alpha_2 := \alpha_1]\, A_2 \hookrightarrow E}{\forall \alpha_1.\, A_1 <: \forall \alpha_2.\, A_2 \hookrightarrow \lambda(f{:}|\forall \alpha_1.\, A_1|).\, \Lambda \alpha.\, E\ (f\ \alpha)}\ \textsc{Sub\_Forall} \qquad \frac{A_1 <: A_2 \hookrightarrow E_1 \qquad A_1 <: A_3 \hookrightarrow E_2}{A_1 <: A_2 \& A_3 \hookrightarrow \lambda(x{:}|A_1|).\, (E_1\ x, E_2\ x)}\ \textsc{Sub\_Inter}$$

$$\frac{A_1 <: A_3 \hookrightarrow E}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda(x{:}|A_1 \& A_2|).\, E\ (\mathsf{proj}_1\, x)}\ \textsc{Sub\_Inter\_1} \qquad \frac{A_2 <: A_3 \hookrightarrow E}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda(x{:}|A_1 \& A_2|).\, E\ (\mathsf{proj}_2 x)}\ \textsc{Sub\_Inter\_2}$$

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

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha\ \mathrm{OK}}\ \textsc{F\_WF\_Var} \qquad \frac{\Gamma \vdash A\ \mathrm{OK} \qquad \Gamma \vdash B\ \mathrm{OK}}{\Gamma \vdash A \to B\ \mathrm{OK}}\ \textsc{F\_WF\_Fun} \qquad \frac{\Gamma, \alpha \vdash A\ \mathrm{OK}}{\Gamma \vdash \forall \alpha.\, A\ \mathrm{OK}}\ \textsc{F\_WF\_Forall}$$

$$\frac{\Gamma \vdash A\ \mathrm{OK} \qquad \Gamma \vdash B\ \mathrm{OK}}{\Gamma \vdash A \& B\ \mathrm{OK}}\ \textsc{F\_WF\_Inter}$$

2

$$\boxed{\Gamma \vdash e : A \hookrightarrow E}$$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A \hookrightarrow x}\ \textsc{F\_Ty\_Var} \qquad \frac{\Gamma \vdash A\ \mathrm{OK} \qquad \Gamma, x{:}A \vdash e : B \hookrightarrow E}{\Gamma \vdash \lambda(x{:}A).\, e : A \to B \hookrightarrow \lambda(x{:}|A|).\, E}\ \textsc{F\_Ty\_Lam}$$

$$\frac{\Gamma \vdash e_1 : A_1 \to A_2 \hookrightarrow E_1 \qquad \Gamma \vdash e_2 : A_3 \hookrightarrow E_2 \qquad A_3 <: A_1 \hookrightarrow E}{\Gamma \vdash e_1\ e_2 : A_2 \hookrightarrow E_1\ (E\ E_2)}\ \textsc{F\_Ty\_App}$$

$$\frac{\Gamma, \alpha \vdash e : A \hookrightarrow E \qquad \Gamma \vdash B\ \mathrm{OK} \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda \alpha.\, e : \forall \alpha.\, A \hookrightarrow \Lambda \alpha.\, E}\ \textsc{F\_Ty\_BLam} \qquad \frac{\Gamma \vdash e : \forall \alpha.\, B \hookrightarrow E \qquad \Gamma \vdash A\ \mathrm{OK}}{\Gamma \vdash e\ A : [\alpha := A]\, B \hookrightarrow E\ |A|}\ \textsc{F\_Ty\_TApp}$$

$$\frac{\Gamma \vdash e_1 : A \hookrightarrow E_1 \qquad \Gamma \vdash e_2 : B \hookrightarrow E_2}{\Gamma \vdash e_1, , e_2 : A \& B \hookrightarrow (E_1, E_2)}\ \textsc{F\_Ty\_Merge}$$

Figure 2. The type system of $F_{\&}$.

Lemma 3 (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 straighforward induction on the derivation[4]. □

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 $F_{\&}$, one may pass more information to a function than what is required; but not in System F. To account for this difference, in «F_Ty_App», the coercion E from the subtyping relation is applied to the argument. «F_Ty_Merge» straighforwardly translates merges into pairs.

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

Theorem 1 (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 2 (Type safety)**.** *If* e *is a well-typed* $F_{\&}$ *term, then* e *evaluates to some System* F *value* v.

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

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

Figure 3. Target language syntax.

$\boxed{|A| = T}$

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

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

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

Figure 4. Type and context translation.

5. Disjointness and Coherence

Although the system shown in the Section 2 is type-safe, it is not coherent. This section shows how to modify the system presented before so that it guarantees coherence as well as type soundness. The keys aspects are the notion of disjoint intersections, and disjoint quantification for polymorphic types.

5.1 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 $F_{\&}$.

Definition 2 (Disjoint types)**.** Given a type enviroment Γ and two types A and B. The two types are said to be disjoint ($\Gamma \vdash A * B$) if they do not share a common supertype. That is, there does not exist a type C such that $\Gamma \vdash A <: C$ and that $\Gamma \vdash B <: C$: [5]

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

Note that any free variables in A, B or C must be bound in Γ.

[5] The definition of disjointness can also be adapted to type systems with a top type (such as `Object` in many OO languages): Two types A and B are *disjoint* if: the fact that C is a commmon supertype of theirs implies C is the top type.

$$
\begin{array}{lrcl}
\text{Types} & A, B & ::= & \alpha \\
& & | & \bot \\
& & | & A \to B \\
& & | & \forall (\alpha * A).\, B \\
& & | & A \& B \\
\text{Terms} & e & ::= & x \\
& & | & \lambda(x{:}A).\, e \\
& & | & e_1\ e_2 \\
& & | & \Lambda(\alpha * A).\, e \\
& & | & e\ A \\
& & | & e_1, , e_2 \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, \alpha * A \mid \Gamma, x{:}A \\
\text{Syntactic sugar} & \Lambda \alpha.\, e & \equiv & \Lambda(\alpha * \bot).\, e \\
& \forall \alpha.\, A & \equiv & \forall (\alpha * \bot).\, A
\end{array}
$$

Figure 5. Amendments of the rules.

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 $\mathsf{Int} <: \mathsf{Int}$. 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 universal type must be disjoint. But a function type and an intersection type may not be. Consider:

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

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

5.2 Syntax

Figure 5 shows the updated syntax with the changes highlighted. Note how similar the changes are to those needed to extend System F with bounded quantification. First, type variables are now always associated with their disjointness constraints (like $\alpha * A$) in types, terms, and contexts. Second, the bottom type ($\bot$) is introduced so that universal quantification becomes a special case of disjoint quantification: $\Lambda \alpha.\, e$ is really a syntactic sugar for $\Lambda(\alpha * \bot).\, e$. The underlying idea is that any type is disjoint with the bottom type. Note the analogy with bounded quantification, where the top type is the trivial upper bound in bounded quantification, while the bottom type is the trivial disjointness constraint in disjoint quantification.

5.3 Typing

Figure 6 shows modifications to Figure 2 in order to support disjoint intersection types and disjoint quantification. Only new rules or rules that different are shown. Importantly, the disjointness judgement appears in the well-formedness rule for intersection types and the typing rule for merges.

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 BRUNO: hummm, is the rule really enforcing that? do we miss a disjointness coinstraint.

$$\boxed{A \text{ atomic}}$$

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

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

$$\frac{A_1 <: A_3 \hookrightarrow E \qquad A_3 \text{ atomic}}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda(x{:}|A_1 \& A_2|).\, E\ (\mathsf{proj}_1\, x)}\ \textsc{Sub\_Inter\_1} \qquad \frac{A_2 <: A_3 \hookrightarrow E \qquad A_3 \text{ atomic}}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda(x{:}|A_1 \& A_2|).\, E\ (\mathsf{proj}_2\, x)}\ \textsc{Sub\_Inter\_2}$$

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

$$\frac{\Gamma \vdash A \text{ OK} \qquad \Gamma, \alpha * A \vdash B \text{ OK}}{\Gamma \vdash \forall(\alpha * A).\, B \text{ OK}}\ \textsc{F\_WF\_Forall} \qquad \frac{\Gamma \vdash A \text{ OK} \qquad \Gamma \vdash B \text{ OK} \qquad \Gamma \vdash A * B}{\Gamma \vdash A \& B \text{ OK}}\ \textsc{F\_WF\_Inter}$$

$$\boxed{\Gamma \vdash e : A \hookrightarrow E}$$

$$\frac{\Gamma \vdash A \text{ OK} \qquad \Gamma, \alpha * A \vdash e : B \hookrightarrow E \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda(\alpha * A).\, e : \forall(\alpha * A).\, B \hookrightarrow \Lambda\alpha.\, E}\ \textsc{F\_Ty\_BLam} \qquad \frac{\Gamma \vdash e : \forall \alpha * B.\, C \hookrightarrow E \qquad \Gamma \vdash A \text{ OK}}{\Gamma \vdash e\ A : [\alpha := A]\ C \hookrightarrow E\ |A|}\ \textsc{F\_Ty\_TApp}$$

$$\frac{\Gamma \vdash e_1 : A \hookrightarrow E_1 \qquad \Gamma \vdash e_2 : B \hookrightarrow E_2 \qquad \Gamma \vdash A * B}{\Gamma \vdash e_1, , e_2 : A \& B \hookrightarrow (E_1, E_2)}\ \textsc{F\_Ty\_Merge}$$

Figure 6. Affected rules.

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. 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 $\alpha * \mathsf{Int}$, α and Int are disjoint:

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

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

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

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

BRUNO: Some connecting text seems to be missing here.

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

Proof. By induction. □

BRUNO: by induction on what?

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

Proof. By induction on the derivation that leads to $\Gamma \vdash e : A$ and applying Lemma 4 in the case of «F_Ty_TApp». □

5.4 Subtyping

The subtyping rules need some adjustment. An important problem with the subtyping rules in Figure 2 is that the all three rules dealing with intersection types («Sub_Inter_1» and «Sub_Inter_2» and «Sub_Inter») overlap. Unfortunatelly, this means that different coercions may be given when checking the subtyping between two types, depending on which derivation is chosen. This is ultimately the reason for incoherence. There are two important types of overlap:

1. The left decomposition rules for intersections («Sub_Inter_1» and «Sub_Inter_2») overlap with each other.
2. The left decomposition rules for intersections («Sub_Inter_1» and «Sub_Inter_2») overlap with the right decomposition rules for intersections («Sub_Inter»).

Fortunatelly, disjoint intersections (which are enforced by well-formedness) deal with problem 1): only one of the two left decomposition rules can be chosen for a disjoint instersection type. Since the two types in the intersection are disjoint it is impossible that both of the preconditions of the left decompositions are satisfied at the same time. More formally, with disjoint intersections, we have the following theorem:

Lemma 6 (Unique subtype contributor)**.** *If* $\Gamma \vdash A_1 \& A_2 <: B$, *where* $A_1 \& A_2$ *and* B *are well-formed types, then it is not possible that the following holds at the same time:*

1. $\Gamma \vdash A_1 <: B$
2. $\Gamma \vdash A_2 <: B$

BRUNO: missing proof sketch

Unfortunatelly, disjoint intersections alone are insufficient to deal with problem 2). In order to deal with problem 2), we introduce a distinction between types, and atomic types.

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

$$A \text{ atomic}$$

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 intersection type, then the only rule that can be applied is «SUB_INTER». 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. Consequentely, the coercion of a subtyping relation $A <: B$ is uniquely determined. This fact is captured by the following lemma:

Lemma 7 (Unique coercion)**.** *If* $\Gamma \vdash A <: B \hookrightarrow E_1$ *and* $\Gamma \vdash A <: B \hookrightarrow E_2$*, where* A *and* B *are well-formed types, then* $E_1 \equiv E_2$.

BRUNO: proof sketch

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

Theorem 3. *If* $A_1 <: A_3$*, then* $A_1 \& A_2 <: A_3$.

Theorem 4. *If* $A_2 <: A_3$*, then* $A_1 \& A_2 <: A_3$.

BRUNO: missing proof sketch

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 judgement in the old systems. GEORGE: Explain why the proof shows this.

GEORGE: Note that $\bot$ does not participate in subtyping and why (because the empty set intersecting the empty set is still empty).

GEORGE: What's the variance of the disjoint constraint? C.f. bounded polymorphism.

GEORGE: Two points are being made here: 1) nondisjoint intersections, 2) atomic types. Show an offending example for each?

5.5 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 «F_TY_APP» (copied below) still needs special attention since we need to show that the generated coercion E is unique.

$$\frac{\Gamma \vdash e_1 : A_1 \to A_2 \hookrightarrow E_1 \qquad \Gamma \vdash e_2 : A_3 \hookrightarrow E_2 \qquad A_3 <: A_1 \hookrightarrow E}{\Gamma \vdash e_1\ e_2 : A_2 \hookrightarrow E_1\ (E\ E_2)}\ \text{F\_TY\_APP}$$

Luckily, by Lemma 5, we know that typing judgements give well-formed types, and thus $\Gamma \vdash A_1$ OK and $\Gamma \vdash A_3$ OK. Therefore we are able to apply Lemma 7 and conclude that E is unique. □

6. Algorithmic Disjointness

Now that we have obtained a specification for disjointness, but the definition involves an existence problem. How can we implement it? One possibility is bidirectional subtyping, that is, we say two types, A and B, are disjoint if neither $A <: B$ nor $B <: A$. However, this implementation is wrong. For example, Int&String and String&Char are not disjoint by specification since String is their common supertype. Yet by the implementation they are, since neither of them is a subtype of the other. BRUNO: You need a concrete code example to make this point. Hence the algorithmic rules are more nuanced. For now, it is enough to treat the disjoint judgement $\Gamma \vdash A * B$ as oracle and we will coome back to this topic in the next section.

As promised, in this section we present the set of rules for determining whether two types are disjoint, and more importantly, how to derive them from the definition we have in the previous sections. The derived set of rules for disjointness is proved to be sound and complete with respect to the definition.

6.1 Derivation

In this subsection, we illustrate how to derive the algorithmic disjointness rules by showing a detailed example for functions. For the ease of discussion, first we introduce some notations.

Definition 3 (Set of common supertypes)**.** For any two types A and B, we can denote the *set of their common supertypes* by

$$\Uparrow (A, B)$$

In other words, a type $C \in \Uparrow (A, B)$ if and only if $A <: C$ and $B <: C$.

Example 6.1. $\Uparrow (\mathsf{Int}, \mathsf{Char})$ is empty, since Int and Char share no common supertype.

Parellel to the notion of the set of common supertypes is the notion of the set of common subtypes.

Definition 4 (Set of common subtypes)**.** For any two types A and B, we can denote the *set of their common subtypes* by

$$\Downarrow (A, B)$$

In other words, a type $C \in \Downarrow (A, B)$ if and only if $C <: A$ and $C <: B$.

Example 6.2. $\Downarrow (\mathsf{Int}, \mathsf{Char})$ is an infinite set which contains Int&Char, Char&Int, (Int&Bool)&Char and so on. But the type Bool is not inside, since it is not a subtype of Int.

Shorthand notation For brevity, we will use

$$\mathcal{A} \to \mathcal{B}$$

as a shorthand for the *set* of types of the form $A \to B$, where $A \in \mathcal{A}$ and $B \in \mathcal{B}$. The same shorthand applies to all other constructors of types, in addition to functions, as well. As a simple example,

$$\{\mathsf{Int}, \mathsf{String}\} \to \{\mathsf{Int}, \mathsf{Char}\}$$

is a shorthand for

$$\{\mathsf{Int} \to \mathsf{Int}, \mathsf{Int} \to \mathsf{Char}, \mathsf{String} \to \mathsf{Int}, \mathsf{String} \to \mathsf{Char}\}$$

Recall that we say two types A and B are disjoint if they do not share a common supertype. Therefore, determining if two types A and B are disjoint is the same as determining if $\Uparrow (A, B)$ is empty.

Determining disjointness of functions Now let's dive into the case where both A and B are functions and consider how to compute $\Uparrow (A_1 \to A_2, B_1 \to B_2)$. By the subtyping rules, the supertype of a function must also be a function.GEORGE: Nah... only after normalizatoin. If not, it can also be & Let $C_1 \to C_2$ be a common supertype of $A_1 \to A_2$ and $B_1 \to B_2$. Then it must satisfy the following:

$$\frac{C_1 <: A_1 \qquad A_2 <: C_2}{A_1 \to A_2 <: C_1 \to C_2} \qquad \frac{C_1 <: B_1 \qquad B_2 <: C_2}{B_1 \to B_2 <: C_1 \to C_2}$$

From which we see that C_1 is a common subtype of A_1 and B_1 and that C_2 is a common supertype of A_2 and B_2. Therefore, we can wrtie:

$$\Uparrow (A_1 \to A_2, B_1 \to B_2) = \Downarrow (A_1, B_1) \to \Uparrow (A_2, B_2)$$

By definition, $\Downarrow (A_1, B_1) \rightarrow \Uparrow (A_2, B_2)$ is not empty if and only if both $\Downarrow (A_1, B_1)$ and $\Uparrow (A_2, B_2)$ is nonempty. However, note that with intersection types, $\Downarrow (A_1, B_1)$ is always nonempty because $A_1 \& B_1$ belongs to $\Downarrow (A_1, B_1)$. Therefore, the problem of determining if $\Uparrow (A_1 \rightarrow A_2, B_1 \rightarrow B_2)$ is empty reduces to the problem of determining if $\Uparrow (B_1 \rightarrow B_2)$ is empty, which is, by definition, if B_1 and B_2 are disjoint. Finally, we have derived a rule for functions:

$$\frac{\Gamma \vdash A_2 *_i B_2}{\Gamma \vdash A_1 \rightarrow A_2 *_i B_1 \rightarrow B_2} \text{ DIS\_FUN}$$

The analysis needed for determining if types with other constructors are disjoint is similar. Below are the major results of the recursive definitions for $\Uparrow$:

$$\begin{aligned} \Uparrow (A_1 \rightarrow A_2, B_1 \rightarrow B_2) &= \Downarrow (A_1, B_1) \rightarrow \Uparrow (A_2, B_2) \\ \Uparrow (A_1 \& A_2, B) &= \Uparrow (A_1, B) \cup \Uparrow (A_1, B) \\ \Uparrow (A, B_1 \& B_2) &= \Uparrow (A, B_1) \cup \Uparrow (A, B_2) \end{aligned}$$

GEORGE: Missing the forall case. But we're just going to drop the formulae.

6.2 Rules

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

Main judgement The judgement $\Gamma \vdash A *_i B$ says two types A and B are disjoint in a context Γ. «DIS_VAR» is the base rule and «DIS_SYM» is its twin rule. «DIS_INTER_1» and «DIS_INTER_2» inductively distribute the relation itself over the intersection constructor (&). «DIS_FUN» is the most interesting rule—it says two function types are disjoint if and only if their return types are disjoint (regardless of their parameter types!). For example, consider the two function types:

$$\mathsf{Int} \rightarrow \mathsf{String} \qquad \mathsf{Bool} \rightarrow \mathsf{String}$$

Even though their parameter types are disjoint, we are still able to think of a type which is a supertype for both of them. For example, $\mathsf{Int\&Bool} \rightarrow \mathsf{String}$. Therefore, by definition, the two function types are not disjoint. The lesson from this example is that the parameter types of two function types do not have a voice in determining whether that two function types are disjoint or not, because in our system given any two types there exists a subtype of the both.

«DIS_AXIOM» says two types are considered disjoint if they are judged to be disjoint by the axiom rules, which is explained below.

Axioms Up till now, the rules of $\Gamma \vdash A *_i B$ have only taken care of two types of the same constructors. But how can be the fact that Int and $\mathsf{Int} \rightarrow \mathsf{Int}$ are disjoint be decided? That is exactly the place where the judgement $A *_{ax} B$ comes in handy. It provides the axioms for disjointness. What is captured by the set of rules is that $A *_{ax} B$ holds for all two types of different constructors unless any of them is an intersection type. That is because for example, $\mathsf{Int\&(Char} \rightarrow \mathsf{Char)}$ and $\mathsf{Char} \rightarrow \mathsf{Char}$ have different constructors and yet are not disjoint. GEORGE: Can you decide that $\bot * \mathsf{Int\&Char}$?

6.3 Metatheory

The algorithmic rules for disjointness is sound and complete.

Lemma 8. *Symmetry of disjointness If* $\Gamma \vdash A * B$*, then* $\Gamma \vdash B * A$.

Theorem 6. *If* $\Gamma \vdash A * C$ *and* $\Gamma \vdash B * C$*, then* $\Gamma \vdash A \& B * C$.

Lemma 9. *If* $A_1 \rightarrow A_2 <: D$ *and* $B_1 \rightarrow B_2 <: D$*, then there exists a* C *such that* $A_2 <: C$ *and* $B_2 <: C$.

Theorem 7 (Soundness of algorithmic disjointness)**.** *For any two types* A *and* B*,* $\Gamma \vdash A *_i B$ *implies* $\Gamma \vdash A * B$.

Proof. By induction on the derivation of $\Gamma \vdash A *_i B$. □

Theorem 8 (Completeness of algorithmic disjointness)**.** *For any two types* A*,* B*,* $\Gamma \vdash A * B$ *implies* $\Gamma \vdash A *_i B$.

Proof. By induction on the derivation of $\Gamma \vdash A * B$. □

7. Discussion

7.1 Union Types

Given this definiton, now may be the right time to take a short digression to consider union types. If a type system ever contains union types (the counterpart of intersection types), with the following standard subtyping rules,

$$\frac{}{A <: A|B} \text{ UNION\_1} \qquad \frac{}{B <: A|B} \text{ UNION\_2}$$

then no two types A and B can ever be disjoint, since there always exists the type $A|B$, which is their common supertype. This serve as the motivation that our system does not permit union types. BRUNO: I wouldn't say this is a motivation: it sounds like we caould not support union types, when I think this is not true. For example we could say something like: there does not exist an *atomic* C ...

7.2 Normalising Types

Since the order of the two types in a binary intersection does not matter, we may normalise them to avoid unnecessary coercions.

7.3 Implementation

We implemented the core functionalities of the $F_{\&}$ as part of a JVM-based compiler. The implementation supports record update instead of restriction as a primitive; however the former is formalized with the same underlying idea of elaborating records. Based on the type system of $F_{\&}$, we 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 multi-field records, mutually recursive let bindings, type aliases, algebraic data types, pattern matching, and first-class modules that are encoded with letrec and records.

Relevant to this paper are the three phases in the compiler, which collectively turn source programs into System F:

1. A *typechecking* phase that checks the usage of $F_{\&}$ features and other source language features against an abstract syntax tree that follows the source syntax.
2. A *desugaring* phase that translates well-typed source terms into $F_{\&}$ terms. Source-level features such as multi-field records, type aliases are removed at this phase. The resulting program is just an $F_{\&}$ term extended with some other constructs necessary for code generation.
3. A *translation* phase that turns well-typed $F_{\&}$ terms into System F ones.

Phase 3 is what we have formalized in this paper.

Removing identity functions. Our translation inserts identity functions whenever subtyping or record operation occurs, which could mean notable run-time overhead. But in practice this is not an issue. In the current implementation, we introduced a partial evaluator with three simple rewriting rules to eliminate the redundant identity functions as another compiler phase after the translation. In another version of our implementation, partial evaluation is

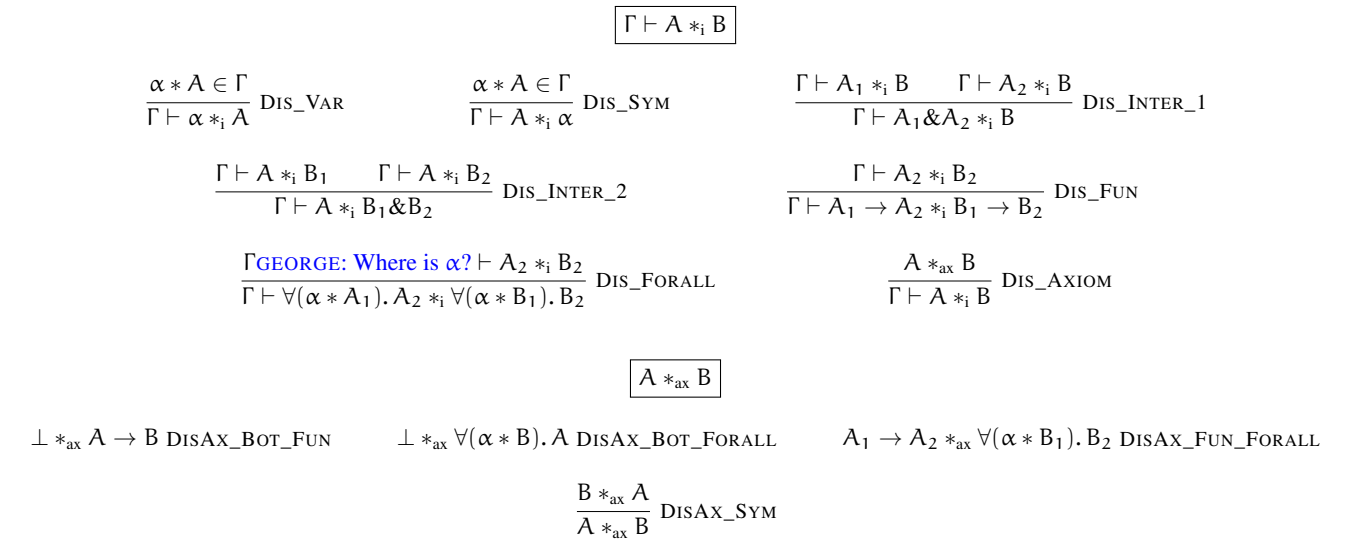

Figure 7. Algorithmic Disjointness.

weaved into the process of translation so that the unwanted identity functions are not introduced during the translation.

8. Related Work

Intersection types with polymorphism. Our type system combines intersection types and parametric polymorphism. Closest to us is Pierce's work [25] on a prototype compiler for a language with both intersection types, union types, and parametric polymorphism. Similarly to $F_{\&}$ in his system universal quantifiers do not support bounded quantification. However Pierce did not try to prove any meta-theoretical results and his calculus does not have a merge operator. Pierce also studied a system where both intersection types and bounded polymorphism are present in his Ph.D. dissertation [26] and a 1997 report [27]. Going in the direction of higher kinds, Compagnoni and Pierce [8] added intersection types to System F_ω and used the new calculus, $F^\omega_\wedge$, to model multiple inheritance. In their system, types include the construct of intersection of types of the same kind K. Davies and Pfenning [12] 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. There have been attempts to provide a foundational calculus for Scala that incorporates intersection types [1, 2]. Although the minimal Scala-like calculus does not natively support parametric polymorphism, it is possible to encode parametric polymorphism with abstract type members. Thus it can be argued that this calculus also supports intersection types and parametric polymorphism. However, the type-soundness of a minimal Scala-like calculus with intersection types and parametric polymorphism is not yet proven. Recently, some form of intersection types has been adopted in object-oriented languages such as Scala, Ceylon, and Grace. Generally speaking, the most significant difference to $F_{\&}$ is that in all previous systems there is no explicit introduction construct like our merge operator. As shown in Section 2, this feature is pivotal in supporting modularity and extensibility because it allows dynamic composition of values.

Other type systems with intersection types. Intersection types date back to as early as Coppo et al. [9]. As emphasized throughout the paper our work is inspired by Dunfield [15]. He described a similar approach to ours: compiling a system with intersection types into ordinary λ-calculus terms. The major difference is that his system does not include parametric polymorphism, while ours does not include unions. Besides, our rules are algorithmic and we formalize a record system. Reynolds invented Forsythe [30] in the 1980s. Our merge operator is analogous to his p_1, p_2. As Dunfield has noted, in Forsythe merges can be only used unambiguously. For instance, it is not allowed in Forsythe to merge two functions.

Refinement intersection [11, 14, 18] 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 syntatic subtyping described in this paper, Frisch et al. [19] studied semantic subtyping.

Languages for extensibility. To improve support for extensibility various researchers have proposed new OOP languages or programming mechanisms. It is interesting to note that design patterns such as object algebras or modular visitors provide a considerably different approach to extensibility when compared to some previous proposals for language designs for extensibility. Therefore the requirements in terms of type system features are quite different. One popular approach is *family polymorphism* [16], which allows whole class hierarchies to be captured as a family of classes. Such a family can be later reused to create a derived family with potentially new class members, and additional methods in the existing classes. *Virtual classes* [17] are a concrete realization of this idea, where a container class can hold nested inner *virtual* classes (forming the family of classes). In a subclass of the container class, the inner classes can themselves be *overriden*, which is why they are called virtual. There are many language mechanisms that provide variants of virtual classes or similar mechanisms [3, 20, 21, 32]. The work by Nystrom on *nested intersection* [21] uses a form of intersection types to support the composition of families of classes. Ostermann's *delegation layers* [24] use delegation for doing dynamic composition in a system with virtual classes. This in contrast with most other approaches that use class-based composition, but closer to the dynamic composition that we use in $F_{\&}$.

9. Conclusion and Future Work

We have described a simple type system suitable for extensible designs. The system has a term-level introduction form for intersection types, combines intersection types with parametric polymorphism, and supports extensible records using a lightweight mechanism. We prove that the translation is type-preserving and the language is type-safe.

There are various avenues for future work. On the one hand we are interested in creating a source language where extensible designs such as object algebras or modular visitors are supported by proper language features. On the other hand we would like to explore extending our structural type system with nominal subtyping to allow more familiar programming experience.

References

[1] N. Amin, A. Moors, and M. Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, 2012.

[2] N. Amin, T. Rompf, and M. Odersky. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Transactions on aspect-oriented software development i. chapter An Overview of Caesarj. 2006.

[4] C. Boehm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] R. H. Bruno C. d. S. Oliveira and A. Loeh. Extensible and modular generics for the masses. In H. Nilsson, editor, *Trends in Functional Programming*. 2006.

[6] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system f with subtyping. *Inf. Comput.*, 109(1-2), Feb. 1994.

[7] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5), 2009.

[8] A. B. Compagnoni and B. C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 1996.

[9] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 1981.

[10] B. C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In *23rd European Conference on Object Oriented Programming (ECOOP)*, 2009.

[11] R. Davies. *Practical refinement-type checking*. PhD thesis, University of Western Australia, 2005.

[12] R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, 2000.

[13] B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. 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.

[14] J. Dunfield. Refined typechecking with stardust. In *Proceedings of the 2007 workshop on Programming languages meets program verification*. ACM, 2007.

[15] J. Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 2014.

[16] E. Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, 2001.

[17] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. *POPL 2006*.

[18] T. Freeman and F. Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, 1991.

[19] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)*, 2008.

[20] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fasioned java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, 2001.

[21] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested intersection for scalable software composition. In *In Proc. 2006 OOPSLA*.

[22] B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses. In *ECOOP 2012–Object-Oriented Programming*. 2012.

[23] B. C. d. S. Oliveira, T. Van Der Storm, A. Loh, and W. R. Cook. Feature-oriented programming with object algebras. In *ECOOP 2013–Object-Oriented Programming*. 2013.

[24] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, 2002.

[25] B. C. Pierce. Programming with intersection types, union types, and polymorphism. 1991.

[26] B. C. Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 1991.

[27] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 1997.

[28] T. Rendel, J. I. Brachthäuser, and K. Ostermann. From object algebras to attribute grammars. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, 2014.

[29] J. C. Reynolds. The coherence of languages with intersection types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, TACS '91, 1991.

[30] J. C. Reynolds. *Design of the programming language Forsythe*. 1997.

[31] C. Schwaab and J. G. Siek. Modular type-safety proofs in agda. In *Proceedings of the 7th Workshop on Programming languages meets program verification (PLPV)*, 2013.

[32] Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, 1998.

[33] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4), July 2008.

[34] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

[35] M. Torgersen. The Expression Problem Revisited. In M. Odersky, editor, *Proc. of the 18th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, 2004.

[36] P. Wadler. The expression problem. *Java-genericity mailing list*, 1998.

[37] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *FOOL*, 2005.

A. Target Type System

$\boxed{G \vdash T \text{ OK}}$

$$\frac{\mathsf{ftv}(T) \in G}{G \vdash T \text{ OK}} \text{ TGT\_WF\_FV}$$

$\boxed{G \vdash E : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \text{ TGT\_TY\_VAR} \qquad \frac{\Gamma \vdash T \text{ OK} \qquad \Gamma, x{:}T \vdash E : T_2}{\Gamma \vdash \lambda(x{:}T_1).\, E : T_1 \to T_2} \text{ TGT\_TY\_LAM} \qquad \frac{\Gamma \vdash E_1 : T_1 \to T_2 \qquad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1\ E_2 : T_2} \text{ TGT\_TY\_APP}$$

$$\frac{\Gamma, \alpha \vdash E : T}{\Gamma \vdash \Lambda\alpha.\, E : \forall\alpha.\, T} \text{ TGT\_TY\_BLAM} \qquad \frac{\Gamma \vdash E : \forall\alpha.\, T_1 \qquad \Gamma \vdash T \text{ OK}}{\Gamma \vdash E\ T : [\alpha := T]\, T_1} \text{ TGT\_TY\_TAPP} \qquad \frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : (T_1, T_2)} \text{ TGT\_TY\_PAIR}$$

$$\frac{\Gamma \vdash E : (T_1, T_2)}{\Gamma \vdash \mathsf{proj}_1 E : T_1} \text{ TGT\_TY\_PROJ\_1} \qquad \frac{\Gamma \vdash E : (T_1, T_2)}{\Gamma \vdash \mathsf{proj}_2 E : T_2} \text{ TGT\_TY\_PROJ\_2}$$

Figure 8. Target type system.

B. Proofs

B.1 Type Safety of $F_{\&}$

We show the type safety of the version of $F_{\&}$ without coherence.

Lemma 3 (Subtyping rules produce type-correct coercions)**.** *If* $A_1 <: A_2 \hookrightarrow E$, *then* $\cdot \vdash E : |A_1| \to |A_2|$.

Proof. By structural induction of the derivation.

- **Case**

  $$\frac{}{\Gamma \vdash \alpha <: \alpha \hookrightarrow \lambda(x{:}|\alpha|).\, x} \text{ SUB\_VAR}$$

  By «TGT_TY_VAR» and «TGT_TY_LAM», $\cdot \vdash \lambda(x{:}|\alpha|).\, x : \alpha \to \alpha$.
- **Case**

  $$\frac{\Gamma \vdash B_1 <: A_1 \hookrightarrow E_1 \qquad \Gamma \vdash A_2 <: B_2 \hookrightarrow E_2}{\Gamma \vdash A_1 \to A_2 <: B_1 \to B_2 \hookrightarrow \lambda(f{:}|A_1 \to A_2|).\, \lambda(x{:}|B_1|).\, E_2\ (f\ (E_1\ x))} \text{ SUB\_FUN}$$

  By i.h., $\cdot \vdash E_1 : |A_3| \to |A_1|$ and $\cdot \vdash E_2 : |A_2| \to |A_4|$. By «TGT_TY_VAR», $\cdot, x{:}|A_3| \vdash x : |A_3|$. By premise, $A_3 <: A_1 \hookrightarrow E_1$. By «TGT_TY_APP», $\cdot, x : |A_3| \vdash E_1\ x : |A_1|$. By «TGT_TY_VAR», $\cdot, f : |A_1 \to A_2| \vdash f : |A_1 \to A_2|$. By the definition of $|\cdot|$, $\cdot, f : |A_1 \to A_2| \vdash f : |A_1| \to |A_2|$. By «TGT_TY_APP», $\cdot, f : |A_1 \to A_2|, x : |A_3| \vdash f\ (E_1\ x) : |A_2|$. By «TGT_TY_APP», $\cdot, f{:}|A_1 \to A_2|, x{:}|A_3| \vdash E_2\ (f\ (E_1\ x)) : |A_4|$. By applying «TGT_TY_LAM» twice, $\cdot \vdash \lambda(f{:}|A_1 \to A_2|).\, \lambda(x{:}|A_3|).\, E_2\ (f\ (E_1\ x)) : |A_1 \to A_2| \to |A_3| \to |A_4|$. By the definition of $|\cdot|$, $\cdot \vdash \lambda(f{:}|A_1 \to A_2|).\, \lambda(x{:}|A_3|).\, E_2\ (f\ (E_1\ x)) : |A_1 \to A_2| \to |A_3 \to A_4|$

- **Case**

  $$\frac{A_1 <: [\alpha_2 := \alpha_1]\, A_2 \hookrightarrow E}{\forall\alpha_1.\, A_1 <: \forall\alpha_2.\, A_2 \hookrightarrow \lambda(f{:}|\forall\alpha_1.\, A_1|).\, \Lambda\alpha.\, E\ (f\ \alpha)} \text{ SUB\_FORALL}$$

  By i.h., $\cdot \vdash E : |A_1| \to |[\alpha_2 := \alpha_1]\, A_2|$. By «TGT_TY_VAR», $\cdot, f{:}|\forall\alpha_1.\, A_1| \vdash f : |\forall\alpha_1.\, A_1|$. By the definition of $|\cdot|$, $\cdot, f{:}|\forall\alpha_1.\, A_1| \vdash f : \forall\alpha_1.|A_1|$. By «TGT_TY_VAR» and «TGT_TY_TAPP», $\cdot, f : |\forall\alpha_1.\, A_1|, \alpha \vdash f\ \alpha : [\alpha_1 := \alpha]\, |A_1|$. By «TGT_TY_APP», $\cdot, f : |\forall\alpha_1.\, A_1|, \alpha \vdash C\ (f\ \alpha) : [\alpha_1 := \alpha]\, |[\alpha_2 := \alpha_1]\, A_2|$. By «TGT_TY_BLAM» and substitution GEORGE: Substitution is problematic, $\cdot, f{:}|\forall\alpha_1.\, A_1| \vdash \Lambda\alpha.\, E\ (f\ \alpha) : \forall\alpha_2.|A_2|$. By «TGT_TY_LAM», $\cdot \vdash \lambda(f{:}|\forall\alpha_1.\, A_1|).\, \Lambda\alpha.\, E\ (f\ \alpha) : |\forall\alpha_1.\, A_1| \to \forall\alpha_2.|A_2|$. By the definition of $|\cdot|$, $\cdot \vdash \lambda(f{:}|\forall\alpha_1.\, A_1|).\, \Lambda\alpha.\, E\ (f\ \alpha) : |\forall\alpha_1.\, A_1| \to |\forall\alpha_2.\, A_2|$.

- **Case**

  $$\frac{A_1 <: A_2 \hookrightarrow E_1 \qquad A_1 <: A_3 \hookrightarrow E_2}{A_1 <: A_2 \& A_3 \hookrightarrow \lambda(x{:}|A_1|).\, (E_1\ x, E_2\ x)} \text{ SUB\_INTER}$$

  By «TGT_TY_VAR», $\cdot, x{:}|A_1| \vdash x : |A_1|$. By i.h., $\cdot \vdash E_1 : |A_1| \to |A_2|$. By «TGT_TY_APP» and weakening, $\cdot, x{:}|A_1| \vdash E_1\ x : |A_2|$. Similarly, $\cdot, x : |A_1| \vdash E_2\ x : |A_3|$. By «TGT_TY_PAIR», $\cdot, x : |A_1| \vdash (E_1\ x, E_2\ x) : (|A_2|, |A_3|)$. By the definition of $|\cdot|$,

$\cdot, x{:}|A_1| \vdash (E_1\ x, E_2\ x) : |A_2 \& A_3|$. By «TGT_TY_LAM», $\cdot \vdash \lambda(x{:}|A_1|).\,(E_1\ x, E_2\ x) : |A_1| \to |A_2 \& A_3|$

- **Case**

$$\frac{A_1 <: A_3 \ \hookrightarrow E}{A_1 \& A_2 <: A_3 \ \hookrightarrow \lambda(x{:}|A_1 \& A_2|).\,E\ (\mathsf{proj}_1 x)}\ \textsc{Sub\_Inter\_1}$$

By «TGT_TY_VAR», $\cdot, x{:}|A_1 \& A_2| \vdash x : |A_1 \& A_2|$. By the definition of $|\cdot|$, $\cdot, x{:}|A_1 \& A_2| \vdash x : (|A_1|, |A_2|)$. By «TGT_TY_PROJ_1», $\cdot, x{:}|A_1 \& A_2| \vdash \mathsf{proj}_1 x : |A_1|$. By i.h., $\cdot \vdash E : |A_1| \to |A_3|$. By weakening, $\cdot, x{:}|A_1 \& A_2| \vdash E : |A_1| \to |A_3|$. By «TGT_TY_APP», $\cdot, x{:}|A_1 \& A_2| \vdash E\ (\mathsf{proj}_1 x) : |A_3|$. By «TGT_TY_LAM», $\cdot \vdash \lambda(x{:}|A_1 \& A_2|).\,E\ (\mathsf{proj}_1 x) : |A_1 \& A_2| \to |A_3|$.

- **Case**

$$\frac{A_2 <: A_3 \ \hookrightarrow E}{A_1 \& A_2 <: A_3 \ \hookrightarrow \lambda(x{:}|A_1 \& A_2|).\,E\ (\mathsf{proj}_2 x)}\ \textsc{Sub\_Inter\_2}$$

By symmetry with the above case.

□

Lemma 10 (Preservation of well-formedness)**.** *If* $\Gamma \vdash A$ OK*, then* $|\Gamma| \vdash |A|$ OK.

Proof. By structural induction of the derivation. «TGT_WF_FV» is the only case.

- **Case**

$$\frac{\mathsf{ftv}(T) \in G}{G \vdash T\ \text{OK}}\ \textsc{Tgt\_WF\_FV}$$

By premise, $\mathsf{ftv}(A) \in \Gamma$. By the definition of $|\cdot|$, $\mathsf{ftv}(|A|) \in |\Gamma|$. By «TGT_WF_FV», $|\Gamma| \vdash |A|$ OK.

□

Theorem 1 (Type preservation)**.** *If* $\Gamma \vdash e : A \ \hookrightarrow E$*, then* $|\Gamma| \vdash E : |A|$.

Proof. By structural induction of the derivation.

- **Case**

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A \ \hookrightarrow x}\ \textsc{F\_Ty\_Var}$$

By premise, $(x, A) \in \Gamma$. By the definition of $|\cdot|$, $(x, |A|) \in |\Gamma|$. By «TGT_TY_VAR», $|\Gamma| \vdash x : |A|$.

- **Case**

$$\frac{\Gamma \vdash A\ \text{OK} \qquad \Gamma, x{:}A \vdash e : B \ \hookrightarrow E}{\Gamma \vdash \lambda(x{:}A).\,e : A \to B \ \hookrightarrow \lambda(x{:}|A|).\,E}\ \textsc{F\_Ty\_Lam}$$

By premise, $\Gamma, x{:}A \vdash e : A_1 \ \hookrightarrow E$. By i.h., $|\Gamma, x{:}A| \vdash E : |A_1|$. By the definition of $|\cdot|$, $|\Gamma|, x{:}|A| \vdash E : |A_1|$. By «TGT_TY_LAM», $|\Gamma| \vdash \lambda(x{:}|A|).\,E : |A| \to |A_1|$. By the definition of $|\cdot|$, $|\Gamma| \vdash \lambda(x{:}|A|).\,E : |A \to A_1|$.

- **Case**

$$\frac{\Gamma \vdash e : \forall \alpha.\,B \ \hookrightarrow E \qquad \Gamma \vdash A\ \text{OK}}{\Gamma \vdash e\ A : [\alpha := A]\,B \ \hookrightarrow E\ |A|}\ \textsc{F\_Ty\_TApp}$$

By premise, $\Gamma \vdash e_1 : A_1 \to A_2 \ \hookrightarrow E_1$. By i.h., $|\Gamma| \vdash E_1 : |A_1 \to A_2|$. By premise, $\Gamma \vdash e_2 : A_3 \ \hookrightarrow E_2$. By i.h., $|\Gamma| \vdash E_2 : |A_3|$. By premise, $A_3 <: A_1 \ \hookrightarrow C$. By Lemma 3, $\cdot \vdash C : |A_3| \to |A_1|$. By «TGT_TY_APP» and the definition of $|\cdot|$, $|\Gamma| \vdash E_1\ (C\ E_2) : |A_2|$.

- **Case**

$$\frac{\Gamma, \alpha \vdash e : A \ \hookrightarrow E \qquad \Gamma \vdash B\ \text{OK} \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda\alpha.\,e : \forall \alpha.\,A \ \hookrightarrow \Lambda\alpha.\,E}\ \textsc{F\_Ty\_BLam}$$

By premise, $\Gamma, \alpha \vdash e : A \ \hookrightarrow E$. By i.h., $|\Gamma, \alpha| \vdash E : |A|$. By the definition of $|\cdot|$, $|\Gamma|, \alpha \vdash E : |A|$. By «TGT_TY_BLAM», $|\Gamma| \vdash \Lambda\alpha.\,E : \forall \alpha.\,|A|$. By the definition of $|\cdot|$, $|\Gamma| \vdash \Lambda\alpha.\,E : |\forall \alpha.\,A|$.

- **Case**

$$\frac{\Gamma \vdash e : \forall \alpha.\, B \hookrightarrow E \qquad \Gamma \vdash A \text{ OK}}{\Gamma \vdash e\ A : [\alpha := A]\ B \hookrightarrow E\ |A|}\ \text{F\_TY\_TAPP}$$

By premise, $\Gamma \vdash e : \forall \alpha.\, A_1 \hookrightarrow E$. By i.h., $|\Gamma| \vdash E : |\forall \alpha.\, A_1|$. By the definition of $|\cdot|$, $|\Gamma| \vdash E : \forall \alpha.\, |A_1|$. By premise, $\Gamma \vdash A$ OK. By Lemma 10, $|\Gamma| \vdash |A|$ OK. By «TGT_TY_TAPP», $\Gamma \vdash E\ |A| : [\alpha := |A|]\ |A_1|$. By substitution lemma, $\Gamma \vdash E\ |A| : |[\alpha := A]\ A_1|$.

- **Case**

$$\frac{\Gamma \vdash e_1 : A \hookrightarrow E_1 \qquad \Gamma \vdash e_2 : B \hookrightarrow E_2}{\Gamma \vdash e_1, , e_2 : A \& B \hookrightarrow (E_1, E_2)}\ \text{F\_TY\_MERGE}$$

By premise, $\Gamma \vdash e_1 : A_1 \hookrightarrow E_1$. By i.h., $|\Gamma| \vdash E_1 : |A_1|$. Similar to the above , $|\Gamma| \vdash E_2 : |A_2|$. By «TGT_TY_PAIR», $|\Gamma| \vdash (E_1, E_2) : (|A_1|, |A_2|)$. By the definition of $|\cdot|$, $|\Gamma| \vdash (E_1, E_2) : |A_1 \& A_2|$.

□

B.2 Coherence of $F_{\&}$

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

Proof. By induction.

- Case

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \text{ OK}}\ \text{F\_WF\_VAR}$$

If $C = \alpha$, then $[\alpha := A]\ \alpha = A$. Since $\Gamma \vdash A$ OK, it follows that $\Gamma \vdash [\alpha := A]\ \alpha$ OK; otherwise, let $C = \beta$, where β is a type variable distinct from α. Since $\Gamma, \alpha * B \vdash \beta$ OK and α and β are distinct, β must be in Γ and therefore $\Gamma \vdash \beta$ OK, which is equivalent to $\Gamma \vdash [\alpha := A]\ \beta$ OK.

- Case

$$\frac{\Gamma \vdash A \text{ OK} \qquad \Gamma \vdash B \text{ OK}}{\Gamma \vdash A \to B \text{ OK}}\ \text{F\_WF\_FUN}$$

By straightforwardly applying the i.h and the rule itself.

- Case

$$\frac{}{\Gamma \vdash \bot \text{ OK}}\ \text{F\_WF\_BOT}$$

Trivial.

- Case

$$\frac{\Gamma, \alpha \vdash A \text{ OK}}{\Gamma \vdash \forall \alpha.\, A \text{ OK}}\ \text{F\_WF\_FORALL}$$

By straightforwardly applying the i.h and the rule itself.

- Case

$$\frac{\Gamma \vdash A \text{ OK} \qquad \Gamma \vdash B \text{ OK}}{\Gamma \vdash A \& B \text{ OK}}\ \text{F\_WF\_INTER}$$

Let C in the statement of this lemma be $E_1 \& E_2$. By the condition we know

$$\Gamma, \alpha * B \vdash E_1 \& E_2 \text{ OK}$$

Thus we must have,

$$\Gamma, \alpha * B \vdash E_1 \text{ OK}$$

By the i.h., $\Gamma \vdash [\alpha := A]\ E_1$ OK and similarly $\Gamma \vdash [\alpha := A]\ E_2$ OK. By «F_WF_INTER»,

$$\Gamma \vdash [\alpha := A]\ E_1 \& [\alpha := A]\ E_2 \text{ OK}$$

and hence

$$\Gamma \vdash [\alpha := A]\ (E_1 \& E_2) \text{ OK}$$

□

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

Proof. By induction on the derivation of $\Gamma \vdash e : A$. The case of «F_TY_TAPP» needs special attention

$$\frac{\Gamma \vdash e : \forall \alpha.\, B \hookrightarrow E \qquad \Gamma \vdash A \text{ OK}}{\Gamma \vdash e\ A : [\alpha := A]\ B \hookrightarrow E\ |A|}\ \text{F\_TY\_TAPP}$$

because we need to show that the result of substitution ($[\alpha := A]\ C$) is well-formed, which is evident by Lemma 4. □

Lemma 7 (Unique coercion)**.** *If* $\Gamma \vdash A <: B \hookrightarrow E_1$ *and* $\Gamma \vdash A <: B \hookrightarrow E_2$*, where* A *and* B *are well-formed types, then* $E_1 \equiv E_2$.

Proof. The set of rules for generating coercions is syntax-directed except for the three rules that involve intersection types in the conclusion. Therefore it suffices to show that if well-formed types A and B satisfy $A <: B$, where A or B is an intersection type, then at most one of the three rules applies. In the following, we do a case analysis on the shape of A and B:

- **Case** $A \neq A_1 \& A_2$ and $B = B_1 \& B_2$: Clearly only «SUB_INTER» can apply.
- **Case** $A = A_1 \& A_2$ and $B \neq B_1 \& B_2$: Only two rules can apply, «SUB_INTER_1» and «SUB_INTER_2». Further, by Lemma 6, it is not possible that $A_1 <: B$ and that $A_2 <: B$. Thus we are certain that at most one rule of «SUB_INTER_1» and «SUB_INTER_2» will apply.
- **Case** $A = A_1 \& A_2$ and $B = B_1 \& B_2$[6]: Since B is not atomic, only «SUB_INTER» apply.

□

B.3 Soundness and Completeness of Algorithmic Disjointness

Some ways to rule out the possibility of disjointness.

- If one type is a subtype of the other, then they cannot be disjoint.
- If the set of free type variables of two types overlap, then they cannot be disjoint.

Lemma 11. *If* A *and* B *are two types and* $ftv(A) \cap ftv(B) \neq \varnothing$*, then there does not exist a* Γ *such that* $\Gamma \vdash A * B$.

Theorem 9. *If* $A <: C$*, then* $A \& B <: C$*. If* $B <: C$*, then* $A \& B <: C$.

GEORGE: Add interpretation of the theorem

Proof. By induction on C. If $C \neq E_1 \& E_2$, trivial. If $C = E_1 \& E_2$, Need to show $A <: E_1 \& E_2$ implies $A \& B <: E_1 \& E_2$. By inversion $A <: E_1$ and $A <: E_2$. By the i.h., $A \& B <: E_1$ and $A \& B <: E_2$. By «SUB_INTER», $A \& B <: E_1 \& E_2$. □

Lemma 12. *Symmetry of disjointness*
If $\Gamma \vdash A * B$*, then* $\Gamma \vdash B * A$.

Proof. Trivial by the definition of disjointness. □

Theorem 10. *If* $\Gamma \vdash A * C$ *and* $\Gamma \vdash B * C$*, then* $\Gamma \vdash A \& B * C$.

Lemma 13. *If* $A_1 \to A_2 <: D$ *and* $B_1 \to B_2 <: D$*, then there exists a* C *such that* $A_2 <: C$ *and* $B_2 <: C$.

Proof. By induction on D. □

Theorem 7 (Soundness of algorithmic disjointness)**.** *For any two types* A *and* B*,* $\Gamma \vdash A *_i B$ *implies* $\Gamma \vdash A * B$.

Proof. GEORGE: State the definition of disjointness more clearly in the text.
We need to show that A and B are disjoint in Γ. The proof is by induction on the derivation of $\Gamma \vdash A *_i B$.

- Case
  $$\frac{\alpha * A \in \Gamma}{\Gamma \vdash \alpha *_i A}\ \text{DIS\_VAR}$$
  Suppose the contrary. Then by the definition of disjointness, there exists a type B such that $\Gamma \vdash \alpha <: B$ and $\Gamma \vdash A <: B$. By GEORGE: ftv subtyping lemma, $\alpha \in ftv(B)$. Combining the fact that $\Gamma \vdash A <: B$, we also have $\alpha \in ftv(A)$. But by GEORGE: another lemma, we know $\alpha \notin ftv(A)$. Contradiction.

[6] An example of this case is:

$$(\mathsf{Int} \& \mathsf{Bool}) \& \mathsf{Char} <: \mathsf{Bool} \& \mathsf{Char}$$

- Case
$$\frac{\alpha * A \in \Gamma}{\Gamma \vdash A *_i \alpha}\ \textsc{Dis\_Sym}$$
Similar.

- Case
$$\frac{\Gamma \vdash A_2 *_i B_2}{\Gamma \vdash A_1 \to A_2 *_i B_1 \to B_2}\ \textsc{Dis\_Fun}$$
Suppose the contrary. Then by the definition of disjointness, there exists a type A such that $A_1 \to A_2 <: A$ and that $B_1 \to B_2 <: A$. Therefore by Lemma 13 we know $A_2 <: A$ and $B_2 <: A$. That is, A_2 and B_2 are not disjoint. But by inversion, we must have $\Gamma \vdash A_2 *_i B_2$ and further by the i.h., $\Gamma \vdash A_2 * B_2$, which contradicts with the above consequence that A_2 and B_2 are not disjoint.
GEORGE: May need an extracted lemma here.

- Case
$$\frac{\Gamma\text{GEORGE: Where is } \alpha\text{?} \vdash A_2 *_i B_2}{\Gamma \vdash \forall(\alpha * A_1).\, A_2 *_i \forall(\alpha * B_1).\, B_2}\ \textsc{Dis\_Forall}$$
Suppose the contrary. Then by the definition of disjointness, there exists a type C such that $\forall(\alpha * A_1).\, A_2 <: A$ and that $\forall(\alpha * B_1).\, B_2 <: A$. By the subtyping rules, A must be of the form $\forall(\alpha * C_1).\, C_2$. Thus,
$$\forall(\alpha * A_1).\, A_2 <: \forall(\alpha * C_1).\, C_2$$
and by inversion, $A_2 <: C_2$. Similarly $B_2 <: C_2$. But by inversion we must have $\Gamma \vdash A_2 *_i B_2$ and by the i.h. $\Gamma \vdash A_2 * B_2$. A contradiction similar to the above case arises.
- Case
$$\frac{\Gamma \vdash A_1 *_i B \qquad \Gamma \vdash A_2 *_i B}{\Gamma \vdash A_1 \& A_2 *_i B}\ \textsc{Dis\_Inter\_1}$$
By Lemma 6 and the i.h.

- Case
$$\frac{\Gamma \vdash A *_i B_1 \qquad \Gamma \vdash A *_i B_2}{\Gamma \vdash A *_i B_1 \& B_2}\ \textsc{Dis\_Inter\_2}$$
By Lemma 6, Lemma 12, and the i.h.

- Case
$$\frac{A *_{ax} B}{\Gamma \vdash A *_i B}\ \textsc{Dis\_Axiom}$$
Straightforward.

□

Theorem 8 (Completeness of algorithmic disjointness)**.** *For any two types* A, B, $\Gamma \vdash A * B$ *implies* $\Gamma \vdash A *_i B$.

Proof. Consider the combinations of the shape of A and B. If A and B are of different shape and neither of them is an intersection type, then by the disjoint axioms we already can conclude that $A *_{ax} B$. Thus by «Dis_Axiom», $\Gamma \vdash A *_i B$. The other cases are as follow.

- Case $A = \bot$:
  - Case $B = \bot$: Trivial.
  - Case $B = B_1 \& B_2$: Need to show $\Gamma \vdash \bot * B_1 \& B_2$ implies $\Gamma \vdash \bot *_i B_1 \& B_2$. Apply «Dis_Inter_2» and the resulting conditions can be proved by the i.h.
- Case $A = A_1 \to A_2$:
  - Case $B = B_1 \to B_2$: Need to show $\Gamma \vdash A_1 \to A_2 * B_1 \to B_2$ implies $\Gamma \vdash A_1 \to A_2 *_I B_1 \to B_2$. Apply «Dis_Fun» and the result, $\Gamma \vdash A_2 *_I B_2$, can be proved by the i.h.
  - Case $B = B_1 \& B_2$: Need to show $\Gamma \vdash A_1 \to A_2 * B_1 \& B_2$ implies $\Gamma \vdash A_1 \to A_2 *_I B_1 \& B_2$. Apply «Dis_Inter_2» and the resulting conditions can be proved by the i.h.
- Case $A = \forall(\alpha * A_1).\, A_2$:
  - Case $B = \forall(\alpha * B_1).\, B_2$
  - Case $B = B_1 \& B_2$:
- Case $A = A_1 \& A_2$:
  - Case $B = B_1 \& B_2$: By «Dis_Inter_1» and by the i.h.

□