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 is able to capture various programming language features. While his calculus is type-safe, it is known that it is not *coherent*: different derivations for the same expression can lead to different results. The lack of coherence is an important disadvantage for adoption of his core calculus in implementations of programming languages, as the semantics of the programming language becomes implementation-dependent.

This paper presents λ_i: a core calculus with a variant of *intersection types* and a *merge operator*. The semantics λ_i is both type-safe and coherent. Coherence is achieved by ensuring that intersection types are *disjoint*. Formally, two types are disjoint if they do not share a common supertype. We present a type system that prevents intersection types that are not disjoint, as well as an algorithmic specification to determine whether two types are disjoint. Moreover, we show the applicability of this calculus to express a simple, yet powerful form of dynamically composable traits, paving the way for new designs of object-oriented programming languages.

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 [13] has shown the usefulness of type systems with 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 elegant and type-safe.

While Dunfield's calculus is type-safe, it lacks the property of *coherence*: different derivations for the same expression can lead to different results. The lack of coherence is an important disadvantage for adoption of his core calculus in implementations of programming languages, as the semantics of the programming language becomes implementation dependent. Although Dunfield mentioned the possibility of extending the type system to allow only disjoint intersection types, he did not formalize or further pursue this approach.

This paper presents λ_i: a core calculus with a variant of *intersection types* and a *merge operator*. The semantics of λ_i is both type-safe and coherent. Coherence is achieved by ensuring that intersection types are *disjoint*. Given two types A and B, two types are disjoint ($A * B$)[1] 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 specification is shown to be sound and complete with respect to the formal definition of disjointness. We have mechanized the many of the interesting metatheoretical results using the Coq proof assistant, and the main proof of type-safety and the coherence theorem can be found in the full version of the paper[2].

As an application of λ_i, we show that λ_i can serve as a foundation for a trait-based [21] object-oriented language. Due to the type system of λ_i, the language supports common features of traits such as commutative composition, explicit conflict detection and resolution. In addition, our language also supports instantiating and composing traits dynamically, which cannot be achieved in many existing statically typed object-oriented languages.

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.
- **Dynamically Composable Traits:** As an application of this work, we show how to build a trait-based object-oriented language by leveraging the type system of λ_i. The resulting source language enjoys many features described in the original traits proposal for free.
- **Formalization of λ_i and Proof of Coherence:** An elaboration semantics of λ_i into λ-calculus is given. Type-soundness and

[1] The notation $A * B$ is inspired by the *separating conjunction* construct in Reynolds' separation logic [20].

[2] `https://github.com/zhiyuanshi/intersection/blob/master/esop-16/paper-full.pdf`

coherence are proved and formalized. Several key theorems are mechanically formalized using the Coq theorem prover.

- **Implementation:** An implementation of an extension of λ_i, as well as encodings of the examples presented in the paper, are publicly available online[3].

2. Overview

This section introduces λ_i and its support for intersection types and the merge operator. It then discusses the issue of coherence and shows how the notion of disjoint intersection types achieve a coherent semantics.

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

2.1 Intersection Types and the Merge Operator

Intersection types date back as early as Coppo *et al.*'s work [8]. Since then various researchers have studied intersection types, and some languages have adopted them in one form or another.

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

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

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

Merge Operator. In the previous program we deliberately did not show how to introduce values of an intersection type. There are many variants of intersection types in the literature. Our work follows a particular formulation, where intersection types are introduced by a *merge operator*. As Dunfield [13] 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 λ_i as follows:

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

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

Merge Operator and Pairs. The merge operator is similar to the introduction construct on pairs. An analogous implementation of `x` with pairs would be:

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

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

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

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

2.2 Incoherence

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

```
(fun (x: Int) → x) (1,,2)
```

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

If both results are accepted, we say that the semantics is *incoherent*: there are multiple possible meanings for the same valid program. Dunfield's calculus [13] is incoherent and accepts the program above.

Getting Around Incoherence: Biased Choice. In a real implementation of Dunfield calculus a choice has to be made on which value to compute. For example, one potential option is to always take the left-most value matching the type in the merge. Similarly, one could always take the right-most value matching the type in the merge. Either way, the meaning of a program will depend on a biased implementation choice, which is clearly unsatisfying from the theoretical point of view.

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

This is precisely the approach taken in λ_i. λ_i requires that the two types of in intersection must be *disjoint*. However, although disjointness seems a natural restriction to impose on intersection types, it is not obvious to formalize it. Indeed Dunfield has mentioned disjointness as an option to restore coherence, but he left it for future work due to the non-triviality of the approach.

Searching for a Definition of Disjointness. The first step towards disjoint intersection types is to come up with a definition of disjointness. A first attempt at such definition would be to require that, given two types A and B, both types are not subtypes of each other. Thus, denoting disjointness as $A * B$, 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,,True)
```

[3] https://github.com/hkuplg/fcore/tree/develop/examples/traits

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

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

Nevertheless the following program still leads to incoherence:

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

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

A Proper Definition of Disjointness. In order for two types to be truly 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 not to share a common supertype (excluding $\top$). 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:

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

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

λ_i's type system only accepts programs that use disjoint intersection types. As shown in Section 5 disjoint intersection types will play a crucial rule in guaranteeing that the semantics is coherent.

3. Dynamically Composable Traits

As an application of disjoint intersection types, we show how to model a simple, yet expressive form of dynamically composable traits [21] in λ_i. Traits provide a mechanism of code reuse in object-oriented programming, that can be used as an alternative to multiple inheritance. The interesting aspect about traits is the way conflicts that typically arise in multiple-inheritance are dealt with. Instead of trying to automatically resolve conflicts, traits detect those conflicts and require programmers to explicitly resolve them. This is where the relation to disjoint intersection types comes in: the mechanism to detect incoherence of disjoint intersection types provides us with the mechanism to detect conflicts in traits. We demonstrate various trait features in a simple OO language, and then sketch a straightforward translation from that language to λ_i as a basic form of syntactic sugar.

3.1 Basic Traits

A trait is a collection of related methods that characterizes only a specific perspective of the features of an object. Therefore, compared with programs using inheritance, programs using traits usually have a larger number of small traits rather than fewer but larger classes. Code reuse with traits is easier than with classes, since traits are usually shorter and can be *composed*. In fact, trait composition offers a variety of possibilities: two traits can be "added" together (which is an symmetric operation); methods can be removed from a trait; and trait systems provide conflict detection, etc.

The first example shows basic trait composition. Many social networking sites allow users to "upvote" a comment and the number of upvotes that comment has received is also displayed. We would like to separate the logic for upvotes from comments so that it can be reused in other entities such as posts and sharings. The code below defines a trait, Comment, which contains a single method content.[4]

```
type Comment = { content: () → String } in
trait Comment(content: String) { self: Comment →
  content() = content
} in
```

Next, we create another trait, Up, for tracking the number of upvotes.

```
type Up = { upvotes: () → Int } in
trait Up(upvotes: Int) { self: Up →
  upvotes() = upvotes
} in
```

At this point the reader may wonder why there are duplicate declarations related to Comment and Up. In mainstream OO languages such as Java, a class declaration such as class C { ... } does two things at the same time:

- Declaring a *template* for creating objects;
- Declaring a new *type*.

In contrast, trait declarations in this source language only do the former. Back to our example, the purpose of declaring two types is just to use them for type annotations of the self reference. In the traits literature, a trait usually "requires" some methods and, based on that, "provides" another set of methods. In our examples, the type of self actually denotes what methods are required.

A trait can expect parameters, which become in scope in the entire trait body. For example, the Comment trait is parametrized by content, and the content method does nothing more than returning the eponymous variable.

The origin of self references is always explicit. The Comment trait requires self to be of type Comment, which is defined as a type synonym for a record in the first line. But the name "self" is nothing special. In fact, self is just another parameter after the preceding parameter list, and becomes in scope after the arrow. We could have named it this or even s.

Creating an object is via the **new** keyword, similar to many OO languages, except for one crucial novelty: we can create an object from multiple traits. More precisely, the object is created from the *composition* of those traits. Therefore, we are able to call methods from different traits on a single object. For example, we can create a single object from Comment and Up traits and test its functionality as follows:

```
let comment = new[Comment&Up] (Comment("Have fun!") &
      Up(4))
in println(comment.content(), comment.upvotes())
-- Output: "Have fun!" 4
```

3.2 Traits with Dependencies

The following example shows that a trait can depend on another trait. First we define the type of a point and a trait for a standard point.

```
type Point = {x: () → Int, y: () → Int} in
trait Point(x: Int, y: Int) { self: Point →
  x() = x
  y() = y
} in
```

The norm of a point can be defined as its distance to the origin. We provide two definitions of norm via two traits.

[4] Our source language assumes regular record type, record operations as well the unit type and unit literals.

```
type Norm = { norm: () → Double } in
 trait EuclideanNorm() { self: Point →
  norm() = Math.sqrt(self.x() * self.x() + self.y() *
     self.y())
} in

 trait ManhattanNorm() { self: Point →
  norm() = Math.abs(self.x()) + Math.abs(self.y())
} in
```

Note how in `EuclideanNorm` and `ManhattanNorm` the type of the self-reference is `Point`! This is in contrast to a typical object-oriented language, such as Java, where the self-reference must always be of the same type as the class being defined. It is this functionality that allow us to express dependencies between traits. When the traits `EuclideanNorm` and `ManhattanNorm` are instantiated and composed with some other traits, they must be composed with an implementation of `Point`.

```
println(new[Point&Norm] (Point(3,4) &
     EuclideanNorm()).norm()) --Prints 5
println(new[Point&Norm] (Point(3,4) &
     ManhattanNorm()).norm()) --Prints 7
```

3.3 Mutual Dependencies

The next example, although a little bit contrived, illustrates that when two traits are composed, any two methods in those two traits can refer to each other via the self reference, just as if they were inside the same class.

```
type EvenOdd = {
  even: Int → Bool,
  odd:  Int → Bool
} in
 trait Even() { self: EvenOdd →
  even(n: Int) = if n == 0 then True else self.odd(n -
     1)
} in
 trait Odd() { self: EvenOdd →
  odd(n: Int) = if n == 0 then False else self.even(n -
     1)
} in
new[EvenOdd] (Even() & Odd()).odd(42)
```

When the two traits are composed, conceptually it is as if that a new object were being created on the fly by copying all the definitions inside those two traits. If there is any unresolved conflict, the program will be rejected by the type system.

3.4 Detecting and Resolving Conflicts in Trait Composition

Traits usually supports explicit conflict detection and resolution. In inheritance, one pattern to resolve conflicts is for the subclass to override methods defined in the parent. The trait-based approach analog is excluding a method from a trait. We show how the mechanism can be modeled in λ_i. The following example shows a counter object and how we could extend its behavior so that it supports reset. First we define a `Counter` as a type synonym for a record that contains a `val` method, which returns the current counter value. Next we define a trait `Counter` that contains two methods. The `val` method just returns the value that is bound at the parameter of the trait, and `incr` returns a new counter.

```
type Counter = { val: () → Int } in
 trait Counter(val: Int) { self: Counter →
  val() = val
  incr() = new[Counter] Counter(val + 1)
} in
type Reset = { reset: () → Counter } in
 trait Reset() { self: Counter →
  reset() = new[Counter] Counter(0)
} in
let counter = new[Counter&Reset] (Counter(0) & Reset())
in counter.incr()
```

In the above code, even though `counter` has a reset method, after we call the `incr` method, the resulting object no longer has that. Therefore, naturally we would like to override the `incr` method inside `Reset`.

```
type Counter = { val: () → Int } in
 trait Counter(val: Int) { self: Counter →
  val() = val
  incr() = new[Counter] Counter(val + 1)
} in
 trait Reset() { self: Counter →
  incr() = new[Counter&Reset] (Counter(val + 1) &
     Reset())
  reset() = new[Counter] Counter(0)
} in
let counter = new[Counter&Reset] (Counter(0) & Reset())
in counter.incr()
```

However the modified code should not typecheck according to the specification of traits, since both `Counter` and `Reset` contains a conflicting `incr` method. The code also does not typecheck since it violates the disjoint intersection typing rules of λ_i. The problem is that both `Counter(0)` and `Reset()` provide an implementation of `incr`. The programmer needs to resolve the conflict by excluding the `incr` being overridden using the record exclusion operator[5].

```
 ...
let counter = new (Counter(0) \ incr & Reset())
in counter.incr()
```

3.5 Dynamic Instantiation

One difference with traditional traits or classes is that in our language we are able to compose traits *dynamically* and then instantiate them. This is impossible in traditional OO languages such as Java since classes being instantiated must be known statically. Actually, since traits are just terms, traits are first-class values and can be defined inside a function, passed around or returned just as normal terms. For example, the following function takes a trait, `norm`, with unknown implementation, compose it with the statically known `Point` trait, and then instantiate it. `Trait` is a built-in keyword and `Trait[Norm]` is the type for traits that conform to the `Norm` type.

```
let makePoint (norm: Trait[Norm]) = new[Point & Norm]
     (Point(3,4) & norm)
in  ...
```

3.6 Desugaring

Of course, this whole section will lose its point if the source language cannot be translated to λ_i and checked against the type system of λ_i. A more formal description can be found in the full version of the paper. The idea of trait translation is inspired by the functional mixin semantics [7] using open recursion, which was proposed by Cook in an untyped setting. However, our translation is done context of a statically-typed programming language, which is what provides the ability to *statically* detect conflicts in traits.

Trait Declarations. A trait in the source language is translated into nothing but a normal term in λ_i. For example,

```
 trait Point(x: Int, y: Int) { self: Point →
  x() = x
  y() = self.z()
}
 ...
```

becomes

[5] We use `e\l` to denote the term where the field `l` is removed. For example, `{x = 1, y = 2} \ x` gives `{x = 1}`.

```
let Point (x: Int, y: Int) (self: () → Point) = {
  x = λ(_: ()) → x,
  y = λ(_: ()) → (self ()).z()
} in
```

One difference is that the self reference becomes a thunk and all occurrences of it have been replaced by `self ()`. Moreover the position of the self reference in the parameter list is adjusted. In fact, `self` is not a special keyword. It can have any name, but `self` is a convention.

The body of the trait becomes a record whose labels are the method names. `Point` has type:

```
(Int, Int) → (() → Point) → Point
```

The syntax for construction such as `Point(3,4)` is just function application in λ_i. And note that `Point(3,4)` is of type

```
(() → Point) → Point
```

Therefore it is an open recursive term: the recursive call is passed as an argument to the definition of `Point` itself.

The "new" Keyword. **new** instantiates a trait by taking the fixpoint of its corresponding open term. In fact, **new** is translated as an inlined fixpoint. For example,

```
new[Point] Point(3,4)
```

becomes

```
let rec self : () → Point = λ(_: ()) → Point (3, 4)
    self
in self ()
```

The composition of traits in the source language is desugared using the merge operator. The reason that traits built on λ_i have conflict detection for free is that the merge operator is enforcing that the two terms being merged are disjoint. For example,

```
new[Point3D] (Point(3,4) & Z(5))
```

is turned into

```
let rec self : () → Point3D = λ(_: ()) → (Point (3,
    4) self) ,, (Z 5 self)
in self ()
```

The "Trait" Keyword. The capitalized `Trait` keyword expects a type and is translated into an open type. For example,

```
Trait[Point]
```

becomes

```
(() → Point) → Point
```

4. The λ_i Calculus

This section presents the syntax, subtyping, typing, as well as the (incoherent) semantics of λ_i: a calculus with intersection types and a merge operator. This calculus is similar to Dunfield's calculus [13]. The differences are that, in Dunfield's language, the coercions are source language terms and that function application allows the expression denoting the function to have an intersection type. In contrast, in λ_i coercions are target language terms and function applications do not allow the expression denoting the applied function to have an intersection type. The later difference will play an important role in achieving coherence. Sections 5 and 6 will present the more fundamental contributions of this paper by showing other necessary changes for supporting disjoint intersection types and ensuring coherence.

$$
\begin{array}{llll}
\text{Types} & A, B, C & ::= & \mathtt{Int} \\
 & & \mid & A \to B \\
 & & \mid & \boxed{A \& B} \\
 & & \mid & \top \\
 & & & \\
\text{Terms} & e & ::= & i \\
 & & \mid & x \\
 & & \mid & \lambda(x\!:\!A).\, e \\
 & & \mid & e_1\ e_2 \\
 & & \mid & \boxed{e_1 ,, e_2} \\
 & & \mid & \top \\
 & & & \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x\!:\!A
\end{array}
$$

Figure 1. λ_i syntax.

4.1 Syntax

Figure 1 shows the syntax. The differences to the λ-calculus, 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 function types $A \to B$. $A\&B$ denotes the intersection of types A and B. We also include $\top$ and integer types `Int`.

Terms. Metavariables e range over terms. Terms include standard constructs: variables x; abstraction of terms over variables of a given type $\lambda(x : A).\, e$; and application of terms e_1 to terms e_2, written $e_1\ e_2$. 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 $A\&B$. The term $\top$ corresponds to the only inhabitant of the $\top$ type. In addition, we also include integer literals i.

Contexts. Typing contexts Γ track bound variables x with their type A.

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. The formalization of nonessential features such as records and record operations can be found in the full version of the paper.

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. 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 the reader to the repository associated with the paper[6].

[6] `https://github.com/zhiyuanshi/intersection/tree/master/esop-16`

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

$$\frac{}{\alpha <: \alpha \hookrightarrow \lambda(x{:}|\alpha|).\, x}\ \textsc{Sub-Var}$$

$$\frac{}{\texttt{Int} <: \texttt{Int} \hookrightarrow \lambda(x{:}|\texttt{Int}|).\, x}\ \textsc{Sub-Int}$$

$$\frac{B_1 <: A_1 \hookrightarrow E_1 \qquad A_2 <: B_2 \hookrightarrow E_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2 \hookrightarrow \lambda(f{:}|A_1 \rightarrow A_2|).\, \lambda(x{:}|B_1|).\, E_2\ (f\ (E_1\ x))}\ \textsc{Sub-Fun}$$

$$\frac{A_1 <: A_2 \hookrightarrow E_1 \qquad A_1 <: A_3 \hookrightarrow E_2}{A_1 <: A_2 \& A_3 \hookrightarrow \lambda(x{:}|A_1|).\, (E_1\ x, E_2\ x)}\ \textsc{Sub-Inter}$$

$$\frac{A_1 <: A_3 \hookrightarrow E}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda(x{:}|A_1 \& A_2|).\, E\ (\texttt{proj}_1 x)}\ \textsc{Sub-Inter-1}$$

$$\frac{A_2 <: A_3 \hookrightarrow E}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda(x{:}|A_1 \& A_2|).\, E\ (\texttt{proj}_2 x)}\ \textsc{Sub-Inter-2}$$

$$\frac{}{A <: \top \hookrightarrow \lambda(\_{:}|A|).\, ()}\ \textsc{Sub-Top}$$

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

$$\frac{}{\Gamma \vdash \texttt{Int}}\ \textsc{F-WF-Int} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \rightarrow B}\ \textsc{F-WF-Fun}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}\ \textsc{F-WF-Inter} \qquad \frac{}{\Gamma \vdash \top}\ \textsc{F-WF-Top}$$

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

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A \hookrightarrow x}\ \textsc{F-Ty-Var} \qquad \frac{}{\Gamma \vdash i : \texttt{Int}}\ \textsc{F-Ty-Int}$$

$$\frac{\Gamma \vdash A \qquad \Gamma, x{:}A \vdash e : B \hookrightarrow E}{\Gamma \vdash \lambda(x{:}A).\, e : A \rightarrow B \hookrightarrow \lambda(x{:}|A|).\, E}\ \textsc{F-Ty-Lam}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A_1 \rightarrow A_2 \hookrightarrow E_1 \\ \Gamma \vdash e_2 : A_3 \hookrightarrow E_2 \qquad A_3 <: A_1 \hookrightarrow E\end{array}}{\Gamma \vdash e_1\ e_2 : A_2 \hookrightarrow E_1\ (E\ E_2)}\ \textsc{F-Ty-App}$$

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

$$\frac{}{\Gamma \vdash \top : \top \hookrightarrow ()}\ \textsc{F-Ty-Top}$$

Figure 2. The type system of λ_i.

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. The typing judgment is of the form:

$$\Gamma \vdash e : A$$

It reads: "in the typing context Γ, the term e is of type A". The standard rules are those for variables (F-TY-VAR) and lambda abstractions (F-TY-LAM). (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.

Typing Applications The rule that deserves more attention is the application rule (F-TY-APP). This is where the most important difference between λ_i and Dunfield calculus is. In Dunfield's calculus, if `f` and `g` are both functions expecting an integer, the following program typechecks:

```
(f ,, g) 1
```

In our calculus, however, one needs manually extract the desired function by passing `f ,, g` to a function. For example, if `f : Int → Bool` and `g : Int → Char`, we could select `f` from the intersection as follows:

```
((fun (h : Int → Bool) → h) (f ,, g)) 1
```

The reason for this design choice is that otherwise we could have another source of semantic ambiguity. Consider the following program:

```
(idInt ,, idChar) (1,,'c')
```

where `idInt` and `idChar` are, respectively, identity functions on integers and on characters. Clearly both merges in the program (`(f ,, g)` and `(1,,'c')`) are disjoint. Yet there are two possible outcomes for this program: `1` or `'c'`. Thus application requires a different treatment, if we wish to have coherence. Our design choice is simply to only employ subtyping in the argument of the application, but not on the expression being applied. It should be possible to design a "smarter" application rule, that still avoids the ambiguity problems that would be present in Dunfield's calculus, while at the same time avoiding explicitly extracting functions from intersections. However such rule would necessarily be complicated, and rather ad-hoc. So we opted for simplicity. In any case there is no expressiveness loss (only loss of convinience), as it is always possibly to extract the function component from a merge and then apply it.

4.4 Semantics

We define the dynamic semantics of the call-by-value λ_i by means of a type-directed translation to an extension of λ-calculus with pairs.

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 λ-calculus. The typing rules can be found in the extended version of the paper.

Key Idea of the Translation. This translation turns merges into usual pairs, similar to Dunfield's elaboration approach [13]. For example,

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

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

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

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

Figure 3. Target language syntax.

$\boxed{|A| = T}$

$$
\begin{aligned}
|\texttt{Int}| &= \texttt{Int} \\
|A_1 \rightarrow A_2| &= |A_1| \rightarrow |A_2| \\
|A_1 \& A_2| &= (|A_1|, |A_2|) \\
|\top| &= ()
\end{aligned}
$$

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

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

Figure 4. Type and context translation.

This expression will be translated to

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

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

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

Coercive subtyping. The judgment

$$A_1 <: A_2 \hookrightarrow E$$

extends the subtyping judgment 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 $|A_1| \rightarrow |A_2|$ (by Lemma 3). For example,

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

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

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. Finally, all rules produce type-correct coercions:

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

Proof. By a straightforward induction on the derivation[7]. ☐

The translation judgment. The translation judgment $\Gamma \vdash e : A \hookrightarrow E$ extends the typing judgment with an elaborated term on the right hand side of $\hookrightarrow$. The translation ensures that E has type $|A|$. In λ_i, one may pass more information to a function than what is required; but not in ordinary λ-calculus. To account for this difference, in (F-TY-APP), the coercion E from the subtyping relation is applied to the argument. (F-TY-MERGE) straightforwardly translates merges into pairs.

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

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* λ_i *term, then* e *evaluates to some* λ*-calculus value* v.

Proof. Since we define the dynamic semantics of λ_i in terms of the composition of the type-directed translation and the dynamic semantics of λ-calculus, type safety follows immediately. ☐

5. Disjointness and Coherence

Although the system shown in the previous section is type-safe, it is not coherent, in the sense that one source program can be elaborated into two different target programs. This section shows how to modify it so that it guarantees coherence as well as type soundness. The result is a calculus named λ_i. The key aspect is the notion of disjoint intersection.

5.1 Disjointness

Throughout the paper we already presented an intuitive definition for disjointness. We recall the definition here

Definition 2 (Disjoint types)**.** Two types A and B are said to be disjoint (written $\vdash A * B$) if they do not share a common supertype (excluding top). That is, there does not exist a type C such that $A <: C$, $B <: C$ and $C \neq \top$.

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

For example, `Int` and `Char` are disjoint, if their only common supertype is $\top$. On the other hand, `Int` is not disjoint with itself, because $\texttt{Int} <: \texttt{Int}$. 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 an intersection type may not be disjoint. Consider:

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

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

[7] The proofs of major lemmata and theorems can be found in the full version of the paper.

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

$$A \to B \text{ atomic} \qquad \texttt{Int} \text{ atomic} \qquad \top \text{ atomic}$$

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

$$\frac{A_1 <: A_3 \hookrightarrow E \qquad A_3 \text{ atomic}}{A_1\&A_2 <: A_3 \hookrightarrow \lambda(x{:}|A_1\&A_2|).\, E\ (\texttt{proj}_1 x)}\ \textsc{Sub-Inter-1}$$

$$\frac{A_2 <: A_3 \hookrightarrow E \qquad A_3 \text{ atomic}}{A_1\&A_2 <: A_3 \hookrightarrow \lambda(x{:}|A_1\&A_2|).\, E\ (\texttt{proj}_2 x)}\ \textsc{Sub-Inter-2}$$

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

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad \vdash A * B}{\Gamma \vdash A\&B}\ \textsc{F-WF-Inter}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \hookrightarrow E_1 \\ \Gamma \vdash e_2 : B \hookrightarrow E_2 \qquad \vdash A * B\end{array}}{\Gamma \vdash e_1,, e_2 : A\&B \hookrightarrow (E_1, E_2)}\ \textsc{F-Ty-Merge}$$

Figure 5. Affected rules.

5.2 Typing

Figure 5 shows modifications to Figure 2 in order to support disjoint intersection types. Only new rules or rules that different are shown. Importantly, the disjointness judgment 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. Under the new rules, intersection types such as `Int&Int` are no longer well-formed because the two types are not disjoint.

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.

5.3 Subtyping

The subtyping rules need some adjustment. An important problem with the subtyping rules in Figure 2 is that all three rules dealing with intersection types ((Sub-Inter-1) and (Sub-Inter-2) and (Sub-Inter)) overlap. Unfortunately, this means that different coercions may be given when checking the subtyping between two types, depending on which derivation is chosen. This is the ultimate 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).

Fortunately, 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 intersection 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 4 (Unique subtype contributor)**.** *If* $A_1\&A_2 <: B$*, 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. $A_1 <: B$
2. $A_2 <: B$

Unfortunately, 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 judgment

$$A \text{ atomic}$$

Since types in λ_i are simple, the only atomic types are the function type and integers. But in richer systems, it can also include, for example, record types. 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. Consequently, the coercion of a subtyping relation $A <: B$ is uniquely determined. This fact is captured by the following lemma:

Lemma 5 (Unique coercion)**.** *If* $A <: B \hookrightarrow E_1$ *and* $A <: B \hookrightarrow E_2$*, where* A *and* B *are well-formed types, then* $E_1 \equiv E_2$.

No Loss of Expressiveness. Interestingly, 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* $A_1\&A_2 <: A_3$.

The interpretation of the two theorems is that: even though the premise is made stricter by the atomic condition, we can still derive every subtyping relation which is valid in the unrestricted system.

5.4 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.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A_1 \to A_2 \hookrightarrow E_1 \\ \Gamma \vdash e_2 : A_3 \hookrightarrow E_2 \qquad A_3 <: A_1 \hookrightarrow E\end{array}}{\Gamma \vdash e_1\ e_2 : A_2 \hookrightarrow E_1\ (E\ E_2)}\ \textsc{F-Ty-App}$$

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

$\square$

6. Algorithmic Disjointness

Section 5 presented a type system with disjoint intersection types that is both type-safe and coherent. Unfortunately the type system is not algorithmic because the specification of disjointness does not lend itself to an implementation directly. This is a problem, because we need an algorithm for checking whether two types are disjoint or not in order to implement the type-system.

This section presents the set of rules for determining whether two types are disjoint. The set of rules is algorithmic and an implementation is easily derived from them. Therefore we solve the problem of finding an algorithm to compute disjointness. The derived set of rules for disjointness is proved to be sound and complete with respect to the definition of disjointness in Section 5.

6.1 Algorithmic Rules

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

$$\frac{\vdash A_2 *_i B_2}{\vdash A_1 \to A_2 *_i B_1 \to B_2}\ \textsc{Dis-Fun}$$

$$\frac{\vdash A_1 *_i B \qquad \vdash A_2 *_i B}{\vdash A_1 \& A_2 *_i B}\ \textsc{Dis-Inter-1}$$

$$\frac{\vdash A *_i B_1 \qquad \vdash A *_i B_2}{\vdash A *_i B_1 \& B_2}\ \textsc{Dis-Inter-2} \qquad \frac{A *_{ax} B}{\vdash A *_i B}\ \textsc{Dis-Axiom}$$

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

$$\texttt{Int} *_{ax} A \to B\ \textsc{DisAx-Int-Fun} \qquad \texttt{Int} *_{ax} \top\ \textsc{DisAx-Int-Top}$$

$$\top *_{ax} A \to B\ \textsc{DisAx-Top-Fun} \qquad \frac{B *_{ax} A}{A *_{ax} B}\ \textsc{DisAx-Sym}$$

Figure 6. Algorithmic Disjointness.

The rules for the disjointness judgment are shown in Figure 6, which consists of two judgments.

Main Judgment. The judgment $\vdash A *_i B$ says two types A and B are disjoint.

The rules dealing with intersection types ((Dis-Inter-1) and (Dis-Inter-2)) are quite intuitive. The intuition is that if two types A and B are disjoint to some type C, then their intersection ($A\&B$) is also clearly disjoint to C. The rules capture this intuition by inductively distributing the relation itself over the intersection constructor (&). Although those two rules overlap, the order of applying them in an implementation does not matter as applying either of them will eventually leads to the same conclusion, that is, if two types are disjoint or not.

The rule for functions (Dis-Fun) is more interesting. It says that two function types are disjoint if and only if their return types are disjoint (regardless of their parameter types!). At first this rule may look surprising because the parameter types play no role in the definition of disjointness. To see the reason for this consider the two function types:

$$\texttt{Int} \to \texttt{String} \qquad \texttt{Bool} \to \texttt{String}$$

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, $\texttt{Int\&Bool} \to \texttt{String}$. Therefore, two function types with the same return type are not disjoint. Essentially, due to the contravariance of function types, functions of the form $A \to C$ and $B \to C$ always have a common supertype (for example $A\&B \to C$). The lesson from this example is that the parameter types of two function types do not have any influence in determining whether those two function types are disjoint or not: only the return types matter.

Axioms. Up till now, the rules of $\vdash A *_i B$ have only taken care of two types with the same language constructs. But how can be the fact that `Int` and $\texttt{Int} \to \texttt{Int}$ are disjoint be decided? That is exactly the place where the judgment $A *_{ax} B$ 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 constructs unless any of them is an intersection type.

6.2 Metatheory

The following two theorems together say that the algorithmic disjointness judgment and the definition of disjointness are "equivalent". For detailed proofs, we refer to the Coq code in our repository.

Theorem 6 (Soundness of algorithmic disjointness)**.** *For any two types* A *and* B*,* $\vdash A *_i B$ *implies* $\vdash A * B$.

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

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

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

7. Related Work

Coherence. Reynolds invented Forsythe [19] in the 1980s. Our merge operator is analogous to his operator p_1, p_2. Forsythe has a coherent semantics. The result was proved formally by Reynolds [18] in a lambda calculus with intersection types and a merge operator. However the way coherence is ensured is not general enough. He has four different typing rules for the merge operator, each accounting for various possibilities of what the types of the first and second components are. In some cases the meaning of the second component takes precedence (that is, is biased) over the first component. The set of rules is restrictive and it forbids, for instance, the merge of two functions (even when they a provably disjoint). Therefore, Forsythe treatment of coherence is rather ad-hoc. In contrast, disjointness in λ_i has a simple, well-defined specification and it is quite flexible.

Pierce [17] made a comprehensive review of coherence, especially on Curien and Ghelli [9] and Reynolds' methods of proving coherence; but he was not able to prove coherence for his $F_\wedge$ calculus. He introduced a primitive `glue` function as a language extension which corresponds to our merge operator. However, in his system users can "glue" two arbitrary values, which can lead to incoherence.

Our work is largely inspired by Dunfield [13]. He described a similar approach to ours: compiling a system with intersection types and a merge operator into ordinary λ-calculus terms with pairs. One major difference is that our system does not include unions. As acknowledged by Dunfield, his calculus lacks of coherence. He discusses the issue of coherence throughout his paper, mentioning biased choice as an option (albeit a rather unsatisfying one). He also mentioned that the notion of disjoint intersection could be a good way to address the problem, but he did not pursue this option in his work.

Recently, Castagna *et al.* [5] studied an very interesting and coherent calculus that has polymorphism and set-theoretic type connectives (such as intersections, unions, and negations). Unfortunately their calculus does not include a merge operator like ours, which is our major source of difficulty for achieving coherence.

Going in the direction of higher kinds, Compagnoni and Pierce [6] 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 [11] studied the interactions between intersection types and effects in call-by-value languages. And they proposed a "value restriction" for intersection types, similar to value restriction on parametric polymorphism. None of those calculi include a merge operator.

There have been attempts to provide a foundational calculus for Scala that incorporates intersection types [1, 2]. 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 λ_i is that in all previous systems there is no explicit introduction construct like our merge operator.

Other Type Systems with Intersection Types. Refinement intersection [10, 12, 15] is the more conservative approach of adopting intersection types. It increases only the expressiveness of types but not terms. But without a term-level construct like "merge", it is not possible to encode various language features. As an alternative to syntactic subtyping described in this paper, Frisch *et al.* [16] studied semantic subtyping. Semantic subtyping seems to have important advantages over syntactic subtyping. One worthy avenue for future work is to study languages with intersection types and merge operator in a semantic subtyping setting.

λ&

"the solution seems much more powerful than simply forbidding non-overlapping intersections"

No it is not! λ& rejects intersections that are accepted in our system:

Int->Char,Int->Bool

This example violates the well-formedness conditions of λ& (see Section 3.2 of "A Calculus for Overloaded Functions with Subtyping" (1995)). So clearly the conditions imposed by λ& (even for the special case of functions) do not subsume disjoint intersections.

"The problem of coherence with intersection types was studied in the 90's by Castagna in the language λ&"

No it wasn't! Castagna studied the overloading problem for functions and focused only on the "merge" (in our sense) of functions. Our work considers a system with **arbitrary** intersections/merges and tries to present a coherent subset of that. The well-formedness conditions presented in Section 3.2 of the λ& paper cannot be ported to a system with arbitrary intersections, since they assume function types only.

Moreover, although λ& can encode records, **it is unclear how to encode arbitrary merges**. One failed attempt to encode arbitrary merges in λ& is to consider a record with multiple labels of the same name. For example, encoding 1,,'c' as

l=1,l='c'

(where label l is of an isolated atomic type L, see Section 4 of the λ& paper). However, this is also rejected in λ&, since it violates the well- formedness conditions.

Nevertheless λ& is related work and should be discussed.

Traits and Trait Calculi. The seminal paper by Schärli *et al.* introduced the ideas behind traits. In their original paper, they documented an implementation of the trait mechanism in a dynamically typed version of Smalltalk. Fisher and Reppy [14] presented a statically typed calculus that models traits. λ_i is not dedicated to traits; but rather, it supports a source language that models traits. Compared to Fisher and Reppy's calculus, λ_i is more lightweight. For example, self references (as well as other OO-specific constructs) are not built-in λ_i. One reason for the difference is that Fisher and Reppy's calculus supports *classes* in addition to traits, and considers the interaction between them, whereas our object oriented source language is *prototype* (or delegation) based—the mechanism for code reuse is purely traits. Of course, there have been many other formalizations of traits, such as [22]. But most of them are heavyweight and specific to modeling traits and typical class-based models of OOP, and therefore differ from our approach.

Bettini *et al.*'s prototype language, SWRTJ [3] distinguishes, in their terminology, "records" and "traits"—the former contain fields and the latter contain methods. Since we try to model a pure object-oriented language, we have excluded fields, which provide state reuse. In SWRTJ, traits themselves are not meant to be the generator of instances. Instead, another construct, called "classes" are, and make use of traits.

The Scala language also has a notion of "traits". However, unlike what its name suggests, the semantics of trait composition in Scala is more similar to mixins [4]. That is, like traditional mixin semantics, when two traits are composed, Scala attempts to do *implicit resolution of conflicts*. In comparison, the traits modeled in λ_i are intended to model the original trait idea closely, and conflicts must be resolved explicitly. Schärli *et al.* document well the trade-offs between mixins and traits. Aside from that, Scala's traits and our source language's traits have four major differences:

1. Scala's traits cannot be instantiated but only mixed into a class (which can be anonymous), whereas traits in our language can be instantiated directly.
2. Scala's traits cannot take constructor parameters whereas ours can. As in the point example below, our trait is itself a constructor and takes the x- and y-coördinates as parameters:

```
trait Point(x: Int, y: Int) { self: Point →
  x() = x
  y() = y
} in  ...
```

3. Dynamic instantiation is supported in λ_i, but not in Scala. In Scala instantiating an object from a class or traits requires that all classes or traits are statically known.
4. Our model of traits is purely functional, but Scala's traits also support fields, mutable state and abstract types.

8. Conclusion and Future Work

This paper described λ_i: a language that combines intersection types and a merge operator. The language is proved to be type-safe and coherent. To ensure coherence the type system accepts only disjoint intersections. We believe that disjoint intersection types are intuitive, and at the same time expressive. We have shown the applicability of disjoint intersection types to model a simple form of traits.

We implemented the core functionalities of the λ_i as part of a JVM-based compiler. Based on the type system of λ_i, we have built an ML-like source language compiler that offers interoperability with Java (such as object creation and method calls). The source language is loosely based on the more general System F_ω and supports a number of other features, including records, polymorphism, mutually recursive `let` bindings, type aliases, algebraic data types, pattern matching, and first-class modules that are encoded using `letrec` and records.

For the future, we intend to improve our source language and show the power of disjoint intersection types in large case studies. One pressing challenge is to address the intersection between disjoint intersection types and polymorphism. We are also interested in extending our work to systems with a $\top$ type. This will also require an adjustment to the notion of disjoint types. A suitable notion of disjointness between two types A and B in the presence of $\top$ would be to require that the only common supertype of A and B is $\top$. Finally we would like to study the addition of union types. This will also require changes in our notion of disjointness, since with union types there always exists a type $A|B$, which is the common supertype of two types A and B.

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] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. A prototypical java-like language with records and traits. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 129–138. ACM, 2010.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA'90*, 1990.

[5] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types: Part 1: Syntax, semantics, and evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, 2014.

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

[7] W. Cook and J. Palsberg. *A denotational semantics of inheritance and its correctness*, volume 24. ACM, 1989.

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

[9] P.-L. Curienl and G. Ghelli. Coherence of subsumption. In *CAAP'90: 15th Colloquium on Trees in Algebra and Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 431, page 132. Springer Science & Business Media, 1990.

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

[11] 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.

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

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

[14] K. Fisher and J. Reppy. A typed calculus of traits. In *Proceedings of the 11th Workshop on Foundations of Object-oriented Programming*, 2004.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[21] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP 2003–Object-Oriented Programming*, pages 248–274. 2003.

[22] N. Scharli, S. Ducasse, R. Wuyts, A. Black, et al. Traits: The formal model. 2003.