Disjoint Intersection Types

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. BRUNO: Abstract needs to be fixed, disjointness is no longer the absence of 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.BRUNO: Drop sentence about traits and replace by other contributions?

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 [12] 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*. BRUNO: introduction needs to be rewritten at least from this point!

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 [20] 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 lan-

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] The notation $A * B$ is inspired by the *separating conjunction* construct in Reynolds' separation logic [19].

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

guage 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 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].

BRUNO: Records and record operations missing in the appendix.

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 3 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 [7]. Since then various researchers have studied intersection types [**?**], and some languages have adopted them in one form or another[**?**].BRUNO: Reviewer asked for more references here.

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 values of type A and of type B. For instance, consider the following program in λ_i:

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

If a value `x` has type `Int & Bool` then `x` can be used as an integer and as a boolean. Therefore, `x` can be used as an argument to any function that takes an integer as an argument, and any function that take a boolean as an argument. In the program above the functions `succ` and `not` are simple 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*.BRUNO: Perhaps mention other works with the merge operator, including lambda& As Dunfield [12] 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. In λ_i (following Dunfield's notation), the merge of two values v_1 and v_2 is denoted as $v_1,,v_2$. For example, an implementation of `x` is constructed in λ_i as follows:

```
let x : Int & Boolean = 1,,True in ...
```

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, Bool) = (1, True) 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 `succ` and `not` with pairs, we would need to write a program such as:

```
(succ (fst xPair), not (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 & Bool` is found, the compiler generates code that uses `fst` to extract the corresponding value at runtime.

2.2 (In)Coherence

Coherence is a desirable property for the semantics of a programming language. A semantics is said to be coherent if any *valid program* has exactly one meaning [17] (that is, the semantics is not ambiguous). In contrast a semantics is said to be *incoherent* if there are multiple possible meanings for the same valid program.

Incoherence in Dunfield's calculus A problem with Dunfield's calculus [12] is that it is incoherent. 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,,True` is of type `Int&Bool`. In this case, no matter if `1,,True` is used as `Int` or `Bool`, the result of evaluation is always clear. However, with overlapping types, it is not straightforward anymore to see the intended result. For example, what should be the result of the following program, which asks for an integer (using a type annotation) out of a merge of two integers:

```
(1,,2) : Int
```

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

Dunfield's calculus [12] accepts the program above, and it allows that program to result in `1` or `2`. In other words the results of the program are incoherent.BRUNO: rephrase better?

Getting Around Incoherence 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. Dunfield suggests some other possibilities, such as the possibility of restricting typing of merges so that a merge has type `A` only if exactly one branch has type `A`. He also suggested another possibility, which is to allow only for *disjoint* types in an intersection. This is the starting point for us and the approach that we will investigate in this paper.

2.3 Disjoint Intersection Types and their Challenges

λ_i requires that the two types in an intersection to be *disjoint*. Informally saying that two types are disjoint means that the set of values of both types are disjoint. Disjoint intersection types are potentially useful for coherence, since they can rule out ambiguity when looking up a value of a certain type in an intersection. However there are several issues that need to be addressed first in order to design a calculus with disjoint intersection types and that ensures coherence. The key issues and the solutions provided by our work are discussed next. We emphasize that although Dunfield has mentioned disjointness as an option to restore coherence, he has not studied the approach further or addressed the issues discussed next.

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

Simple disjoint intersection types Looking back at the expression `1,,2` in Section 2.2, 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. A natural option is to try to forbid such overlapping values of the same type in a merge. Thus, for atomic types such as `Int` and `Bool`, it is easy to see that disjointness holds when the two types are different. Intersections such as `Int & Bool` and `String & Bool` are clearly disjoint. While an informal, intuitive notion of disjointness is sufficient to see what happens with atomic types, it is less clear of what disjointness means in general.

Formalizing disjointness Clearly a formal notion of disjointness is needed to design a calculus with disjoint intersection types, and to clarify what disjointness means in general. As we shall see the particular notion of disjointness is quite sensitive to the language features that are allowed in a language. Nevertheless, the different notions of disjointness follow the same principle: they are defined in terms of the subtyping relation; and they describe which common supertypes are allowed in order for two types to be considered disjoint.

A first attempt at a definition for disjointness is 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:

$$A * B \equiv A \not<: B \text{ and } B \not<: A$$

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)
```

This merge has two components which are also merges. 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:

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

as both `2` or `3` 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.

In order for two types to be truly disjoint, they must not have any subcomponents sharing the same type. In a simply typed calculus with intersection types (and without a $\top$ type) this can be ensured by requiring the two types not to share a common supertype:

Definition 1 (Simple disjointness)**.** Two types A and B are disjoint (written $A * B$) if there is no type C such that both A and B are subtypes of C:

$$A * B \equiv \nexists C.\ A <: C \text{ and } B <: C$$

This definition of disjointness prevents the problematic merge `(1,,'c'),,(2,,True)`. Since `Int` is a common supertype of both `Int&Char` and `Int&Bool`, those two types are not disjoint, according to this simple notion of disjointness.

The simple definition of disjointness is the basis for the first calculus presented in this paper: a simply typed lambda calculus with intersection (but without a $\top$ type). This variant of λ_i is useful to study many important issues arizing from disjoint intersections, without the additional complications of $\top$. As shown in Section **??**, this definition of disjointness is sufficient to ensure coherence in λ_i.

Disjointness of non-atomic types Equipped with a formal notion of disjointness, we are now ready to see how disjointness works for other, non-atomic types. For example, consider the following intersection types of functions:

1. $(\texttt{Int} \to \texttt{Int}) \,\&\, (\texttt{String} \to \texttt{String})$
2. $(\texttt{String} \to \texttt{Int}) \,\&\, (\texttt{String} \to \texttt{String})$
3. $(\texttt{Int} \to \texttt{String}) \,\&\, (\texttt{String} \to \texttt{String})$

Which of those intersection types are disjoint? It seems reasonable to expect that the first intersection type is disjoint: both the domain and codomain of the two functions in the intersection are different. However, it is less clear whether the two other intersection types are disjoint or not. Looking at definition 3 for further guidance, and the subtyping rule for functions in λ_i (which is standard [**?**]):

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

we can see that the types in the second intersection do not share any common supertypes. Since the target types of the two function types (`Int` and `String`) do not share a common supertype, it is not possible to find a type `C` that is both a common supertype of $(\texttt{String} \to \texttt{Int})$ and $(\texttt{String} \to \texttt{String})$. In contrast, for the third intersection type, it is possible to find a common supertype: $\texttt{String} \,\&\, \texttt{Int} \to \texttt{String}$. The contravariance of argument types in $S \to$ is important here. All that we need in order to find a common supertype between $(\texttt{Int} \to \texttt{String})$ and $(\texttt{String} \to \texttt{String})$ is to find a common subtype between `Int` and `String`. One such common subtype is $\texttt{String} \,\&\, \texttt{Int}$. Preventing the third intersection type ensures that type-based lookups are not ambiguos (and cannot lead to incoherence). If the third intersection type was allowed then the following program:

```
f,,g : String & Int → String
```

where `f` is of type $\texttt{Int} \to \texttt{String}$ and `g` is of type $\texttt{String} \to \texttt{String}$, would be problematic. In this case both `f` or `g` could be selected, potentially leading to very different (and incoherent) results when applied to some argument.

Is disjointness sufficient to ensure coherence? Another question is whether disjoint intersection types are sufficient to ensure coherence. Consider the following example:

```
(succ,,not) (3,,True)
```

Here there are two merges, with the first merge being applied to the second. The first merge contains two functions (`succ` and `not`). The second merge contains two values that can serve as input to the functions. The two merges are disjoint. However what should be the result of this program? Should it be `4` or `False`?

At first this program appears to lead to incoherence, even though it only uses disjoint merges. However, a closer look reveals that there are two possible types for this program: `Int` or `Bool`. Once the type of the program is fixed, there is only one possible result: if the type is `Int` the result of the program is `4`; if the type is `Bool` then the result of the program is `False`. Like other programming language features (for example type classes []), types play a fundamental role in determining the result of a program, and the semantics of the language is not independent from types.

Disjointness is indeed sufficient to ensure coherence. However, for certain programs type annotations are necessary to rule out *type ambiguity*. Similar type ambiguity issues arize in other type-directed mechanisms such as type-classes [].

In our implementation, which uses bi-directional type-checking techniques [], the ambiguity problem is solved by annotating the term being applied. For example:

```
((succ,,not) : Int → Int) (3,,True)
```

is a valid program and results in `4`.BRUNO: improve text?

2.4 Disjoint Intersection Types with $\top$

In the presence of a $\top$ type the simple definition of disjointness is useless: $\top$ is always a common supertype of any two types. Therefore, with the previous definition of disjointness no disjoint intersections could ever be well-formed in the presence of $\top$! Moreover, since $\top$ is not disjoint to any type, it does not make sense to allow its presence in a disjoint intersection type. Adding a $\top$ type requires some adaptations on the notion of disjointness. This paper studies two additional variants of λ_i with $\top$ types. In both variants the definition of disjointness is revised as follows:

Definition 2 ($\top$-Disjointness)**.** Two types A and B are disjoint (written $A * B$) if the following two conditions are satisfied:

1. $\neg A_\top$ and $\neg B_\top$
2. $\forall C$. if $A <: C$ and $B <: C$ then $C_\top$

In the presence of $\top$, instead of requiring that two types do not share any common supertype, we require that the only allowed common supertypes are *top-like* (condition #2). Additionally, it is also required that the two types A and B are not themselves top-like (condition #1). The unary relation $\cdot_\top$ denotes such top-like types. Top-like types obviously include the $\top$ type. However top-like types also include other types which are *syntactically different* from $\top$, but behave like a $\top$ type. For example, $\top$ & $\top$ is syntactically different from $\top$, but it is still a supertype of every other type (including $\top$ itself). The standard subtyping relation for intersection types includes a rule:

$$\frac{A_1 <: A_2 \qquad A_1 <: A_3}{A_1 <: A_2 \& A_3}\ \text{S\&R}$$

The presence of S&R means that both $\top\&\top <: \top$ and $\top <: \top\&\top$ are derivable. In other words, in a calculus like Dunfield's there are infinitely many syntactically different types that behave like a $\top$ type: $\top$, $\top\&\top$, $\top\&\top\&\top$, ...

The notion of $\top$-Disjointness has two benefits. Firstly, and more importantly, $\top$-Disjointness is sufficient to ensure coherence. For example, the following program is valid, and coherent:

```
3,,True : ⊤
```

Even though the types of both components of the merge are a subtype of the type of the program ($\top$), the result of the program is always the *unique* $\top$ value. Secondly, $\top$-Disjointness has the side-effect of excluding other top-like types from the system: the intersection type $\top\&\top$ is not a well-formed disjoint intersection type. In contrast to Dunfield's calculus, λ_i has a unique syntactic $\top$ type.

Functional intersections and top-like types The two variants of λ_i with $\top$ differ slightly on the definition of top-like types. The concrete definition of top-like types is important because it affects what types are allowed in intersections. In the simpler version of λ_i with $\top$, multiple functions cannot coexist in intersections. This is obviously a drawback and would reduce the usefulness of the system. The essential problem is that a simple notion of top-like types is too restrictive. For example consider again the intersection type:

$$(\texttt{String} \to \texttt{Int}) \,\&\, (\texttt{String} \to \texttt{String})$$

According to the simple definition of disjointness, this disjoint intersection type is valid. However according $\top$-Disjointness and a simple definition of top-like types, which accounts only for proper top types, this disjoint intersection type is not valid. The two types have a common supertype which is not a supertype of every type: $\texttt{String} \to \top$. In this case $(\texttt{String} \to \top) <: \top$, but $\top \not<: (\texttt{String} \to \top)$. Therefore the two types do not meet the second condition of $\top$-Disjointness as there exists a common supertype, which is not top-like.

Types	A, B, C	::=	$\texttt{Int}$
		|	$A \to B$
		|	$A \times B$
		|	$A\&B$
Terms	e	::=	i
		|	x
		|	$\lambda x.\, e$
		|	$e_1\ e_2$
		|	(e_1, e_2)
		|	$\texttt{proj}_k e \quad k \in \{1,2\}$
		|	$e_1,, e_2$
Contexts	Γ	::=	$\cdot \mid \Gamma, x{:}A$

Figure 1. λ_i syntax.

Generalizing top-like types In the second variant of λ_i top-likes are defined more liberally and they allow "local" top types, as well as proper top types. BRUNO: finish this

λ_i's type system only accepts programs that use disjoint intersection types. As shown in Section 4 disjoint intersection types will play a crucial rule in guaranteeing that the semantics is coherent.

3. The λ_i Calculus and its Type System

This section presents the syntax, subtyping and typing of λ_i: a calculus with intersection types and a merge operator. This calculus is inspired by Dunfield's calculus [12]. However our calculus does not consider union typesBRUNO: Consider a discussion about union types later in the paper?. Moreover, due to the fact that we are interested only in disjoint intersections, λ_i also has slighty different typing rules than in Dunfield's calculus. Sections 4 and 5 will present the more fundamental contributions of this paper by showing other necessary changes for supporting disjoint intersection types and ensuring coherence. The calculus in this section does not include the $\top$ type, which brings some additional complications. Section **??** presents two variants of λ_i with a $\top$ type, and shows how coherence can be preserved in the presence of $\top$.

3.1 Syntax

Figure 1 shows the syntax. The difference to the λ-calculus (with pairs), highlighted in gray, are intersection types ($A\&B$) at the type-level, and merges ($e_1,, e_2$) at the term level.

BRUNO: Careful double-checking is needed throughout the paper to ensure that we are not using annotatted lambdas. BRUNO: Probably better to discuss all top rules later, in the top section.

Types. Metavariables A, B range over types. Types include function types $A \to B$ and product types $A \times B$. $A\&B$ denotes the intersection of types A and B. We also include integer types $\texttt{Int}$.

Terms. Metavariables e range over terms. Terms include standard constructs: variables x; abstraction of terms over variables $\lambda x.\, e$; application of terms e_1 to terms e_2, written $e_1\ e_2$; pairing of two terms e_1 and e_2, denoted as (e_1, e_2); and both projections of a pair e, written $\texttt{proj}_k e$ (with $k \in \{1,2\}$). 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 $A\&B$. 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.

3.2 Subtyping

BRUNO: Draw boxes around the stuff that is related to disjointness, as this will be shown later. Perhaps there is no need to show the definition of ordinary, here. Talk about it only later in the paper.

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 S $\rightarrow$ says that a function is contravariant in its parameter type and covariant in its return type. The three rules dealing with intersection types are just what one would expect when interpreting types as sets. Under this interpretation, for example, the rule S&R says that if A_1 is both the subset of A_2 and the subset of A_3, then A_1 is also the subset of the intersection of A_2 and A_3.

Note that the notion of ordinary types, which is used in rules S&R1 and S&R2, was introduced by Davies and Pfenning [] to provide an algorithmic version of subtyping. In our system ordinary types are used for a different purpose as well: they play a fundamantal role in ensuring that subtyping produces unique coercions. Section **??** will present a detailed discussion on this.BRUNO: improve text?

Metatheory. The subtyping relation, is known to be reflexive and transitive [].

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

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

3.3 Declarative Type System

The well-formdness of types and typing relation are shown in the middle and bottom of Figure 2, respectively. Importantly, the disjointness judgment, which is highlighted using a box, appears in the well-formedness rule for intersection types (WF&) and the typing rule for merges (T-MERGE). The presence of the disjointness judgement, as well as the use of ordinary types in the subtyping relation, are the most essential differences between our type system and the original type system by Dunfield.

Apart from WF&, the remaining rules for well-formedness are standard. 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 T-VAR; lambda abstractions T-LAM; application T-APP; integer literals T-INT; products T-PROD; and projections (1 and 2) T-PROJ. T-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, as long as the types of the two expressions are disjoint. Finally, T-SUB states that for any types A and B, if $A <: B$ then any expression e with assigned type B can also be assigned the type A.

Different rules for intersections Dunfield's calculus has different rules for intersections. However, his rules make less sense in a system with disjoint intersections. For example, Dunfield's calculus includes the following typing rule, for introducing intersection types:

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

$A \rightarrow B$ ordinary $\qquad$ $A \times B$ ordinary $\qquad$ $\texttt{Int}$ ordinary

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

$$\frac{}{\texttt{Int} <: \texttt{Int} \hookrightarrow \lambda x.\, x}\ S\mathbb{Z}$$

$$\frac{A_1 <: B_1 \hookrightarrow E_1 \qquad A_2 <: B_2 \hookrightarrow E_2}{A_1 \times A_2 <: B_1 \times B_2 \hookrightarrow \lambda p.\, (E_1\ (\texttt{proj}_1 p), E_2\ (\texttt{proj}_2 p))}\ S\times$$

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

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

$$\frac{A_1 <: A_3 \hookrightarrow E \qquad \boxed{A_3 \text{ ordinary}}}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda x.\, E\ (\texttt{proj}_1 x)}\ S\&L_1$$

$$\frac{A_2 <: A_3 \hookrightarrow E \qquad \boxed{A_3 \text{ ordinary}}}{A_1 \& A_2 <: A_3 \hookrightarrow \lambda x.\, E\ (\texttt{proj}_2 x)}\ S\&L_2$$

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

$$\frac{}{\Gamma \vdash \texttt{Int}}\ WF\mathbb{Z} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \rightarrow B}\ WF\rightarrow$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B}\ WF\times \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad \boxed{A * B}}{\Gamma \vdash A\&B}\ WF\&$$

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

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}\ \text{T-VAR} \qquad \frac{\Gamma \vdash A \qquad \Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \rightarrow B}\ \text{T-LAM}$$

$$\frac{}{\Gamma \vdash i : \texttt{Int}}\ \text{T-INT} \qquad \frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}\ \text{T-PROD}$$

$$\frac{\Gamma \vdash e_1 : A_1 \rightarrow A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2}\ \text{T-APP}$$

$$\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B \qquad \boxed{A * B}}{\Gamma \vdash e_1,,e_2 : A\&B}\ \text{T-MERGE}$$

$$\frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \texttt{proj}_k e : A_k}\ \text{T-PROJ} \qquad \frac{\Gamma \vdash e : A \qquad A <: B}{\Gamma \vdash e : B}\ \text{T-SUB}$$

Figure 2. Declarative type system of λ_i.

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

Figure 3. Target language syntax.

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e : B}{\Gamma \vdash e : A\&B}$$

A first reason why such rule would not work in λ_i is that it does not restrict $A\&B$ to be disjoint. Therefore, in the presence of such unrestricted rule it would be possible to create non-disjoint intersections types. It is easy enough to have an additional disjointness restriction, which would ensure the disjointness between A and B. However, then the rule would be pointless, because no derivations of programs could ever be created with such a rule. If A and B are disjoint then, by definition, no programs should ever have types A and B at the same time. In contrast, in λ_i, the rule T-MERGE captures the fact that two disjoint pieces of evidence are needed to create a (disjoint) intersection type.

4. Semantics, Disjointness and Coherence

This section discusses the elaboration semantics of λ_i, and shows a bidirectional type system that guarantees coherence and type soundness. Moreover the bidirectional type system is shown to be sound and complete with respect to the type system presented in Section 3 (provided that terms have some additional type annotations). The coherence theorem presented in this section, relies on two key aspects of the calculus:

- **Uniqueness of subtyping coercions:** the notion of ordinary types, well-formed disjoint intersection types ensures that coercions produced by subtyping relation are unique.
- **No type ambiguity:** the bi-directional type system does not have type-ambiguity. Thus the bi-directional type system infers a unique type for every well-typed program.

4.1 Target of Elaboration

The dynamic semantics of the call-by-value λ_i is defined via a type-directed translation into the simply typed λ-calculus with pair and unit types. 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, and in our Coq proof scripts.BRUNO: should we mention an extended version of the paper?

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.

$\boxed{|A| = T}$

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

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

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

Figure 4. Type and context translation.

4.2 Coercive Subtyping and Coherence

The λ_i calculus uses coercive subtyping, where subtyping derivations produce a coercion that is used to transform values of one type to another type. Our calculus ensures that the coercions produced by subtyping are unique. Unique coercions are fundamental for proving our coherence result of the semantics of λ_i.

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 $|A_1| \to |A_2|$ (by Lemma 3). For example,

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

generates a coercion function with type: $\texttt{Int\&Bool} \to \texttt{Bool}$. Note that, in contrast to Dunfield's elaboration approach, where subtyping produces coercions that are source language terms, in λ_i, coercions are produced directly on the target language.

In S $\to$, we elaborate the subtyping of parameter and return types by η-expanding f to $\lambda x.\, f\ x$, applying E_1 to the argument and E_2 to the result. Rules S&L$_1$, S&L$_2$, and S&R elaborate intersection types. S&R uses both coercions to form a pair. Rules S&L$_1$ and S&L$_2$ reuse the coercion from the premises and create new ones that cater to the changes of the argument type in the conclusions. 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* $\cdot \vdash E : |A_1| \to |A_2|$.

Proof. By a straightforward induction on the derivation[4]. □

Overlapping subtyping rules The key problem with the subtyping rules in Figure 2 is that all three rules dealing with intersection types (S&L$_1$ and S&L$_2$ and S&R) 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:

[4] The proofs of major lemmata and theorems can be found in the full version of the paper.BRUNO: where??

1. The left decomposition rules for intersections ($S\&L_1$ and $S\&L_2$) overlap with each other.
2. The left decomposition rules for intersections ($S\&L_1$ and $S\&L_2$) overlap with the right decomposition rules for intersections S&R.

Well-formedness and disjointness The fact that in λ_i all intersection types are disjoint is useful to deal with problem 1). Recall the definition of disjointness:

Definition 3 (Simple disjointness)**.** Two types A and B are disjoint (written $A * B$) if there is no type C such that both A and B are subtypes of C:

$$A * B \equiv \nexists C.\ A <: C \text{ and } B <: C$$

Disjoint intersections are enforced by well-formedness of types. Since the two types in an intersection are disjoint, it is impossible that both of the preconditions of the left decompositions are satisfied at the same time. Therefore, only one of the two left decomposition rules can be chosen for a disjoint intersection type. 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 ordinary types.

Ordinary Types. Ordinary types are just those which are not intersection types, and are asserted by the judgment

$$A \text{ ordinary}$$

Since types in λ_i are simple, the only ordinary types are the function type and integers. But in richer systems, it can also include, for example, record types or pairs. In the left decomposition rules for intersections we introduce a requirement that A_3 is ordinary. The consequence of this requirement is that when A_3 is an intersection type, then the only rule that can be applied is S&R. With the ordinary 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* $E_1 \equiv E_2$*.*

4.3 Bidirectional Type System with Elaboration

In order to prove the coherence result we first introduce a bidirectional type system, which is closely related to the type system presented in Section 3. The type system is elaborating, producing a term in the target language while performing the typing derivation.

The bidirectional type system is useful for two different reasons. Firstly, the presence of the subsumption rule (T-SUB) makes the type system not syntax directed, which presents a challenge for an implementation. Bidirectional type-checking makes the rules syntax directed again. Secondly, and more importantly, the subsumption rule also creates type ambiguity: the same term can have multiple types. This is problematic because there can be different semantics for a term, depending on the type of the term. Bidirectional type-checking comes to the rescue again, by ensuring that with the additional type annotations only one type is inferred for a term.

Key Idea of the elaboration. The key idea in the elaboration is to turn merges into usual pairs, similar to Dunfield's elaboration approach [12]. For example,

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

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

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

This expression will be translated to

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

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

The elaboration judgments and rules. Figure 5 presents the elaborating bidirectional type system. The bidirectional type system itself is rather standard. The key differences to the type system in Figure 2 are that all :'s are replaced with $\Rightarrow$ or $\Leftarrow$. There are two check-mode rules: T-LAM and T-SUB. The remaining rules are all in the synthesis mode. Moreover there is one additional rule for annotation expressions (T-ANN). The syntax of source terms also needs to be extended with annotation expressions $e : A$. BRUNO: References to bidirectional type-checking needed somewhere?

The two elaboration judgments $\Gamma \vdash e \Rightarrow A \hookrightarrow E$ and $\Gamma \vdash e \Leftarrow A \hookrightarrow E$ extends the usal typing judgments with an elaborated term on the right hand side of the arrows. The elaboration ensures that E has type $|A|$. Noteworthy are the T-MERGE and T-SUB rules. The T-MERGE straightforwardly translates merges into pairs. The T-SUB accounts for the type coercions arizing from subtyping. The additional coercions are necessary to ensure that the target terms are correctly typed, since the target language lacks subtyping.

Type-safety The type-directed elaboration 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. □

Soundness and Compleness The declarative type system presented in Figure 2 is closely related to the bidirectional type system in Figure 5. We can prove that the bidirectional type system is sound and complete with respect to the declarative specification, modulo some additional type-annotations. We will use the definition of erasure, as shown in Figure 6.

Theorem 3 (Soundness of bidirectional type-checking)**.** *If* $\Gamma \vdash e \Rightarrow A \hookrightarrow E$*, then* $\Gamma \vdash \lfloor e \rfloor : A \hookrightarrow E'$*.*

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

Theorem 4 (Completeness of bidirectional type-checking)**.** *If* $\Gamma \vdash e : A \hookrightarrow E'$*, then* $\Gamma \vdash e' \Rightarrow A \hookrightarrow E$*, where* $\lfloor e' \rfloor = e$*.*

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

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

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash x \Rightarrow A \hookrightarrow x}\ \text{T-VAR} \qquad \frac{}{\Gamma \vdash i \Rightarrow \texttt{Int} \hookrightarrow i}\ \text{T-INT}$$

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

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

$$\frac{\Gamma \vdash e \Rightarrow A_1 \times A_2 \hookrightarrow E}{\Gamma \vdash \texttt{proj}_k e \Rightarrow A_k \hookrightarrow \texttt{proj}_k E}\ \text{T-PROJ}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow A \hookrightarrow E_1 \\ \Gamma \vdash e_2 \Rightarrow B \hookrightarrow E_2 \qquad \boxed{A * B}\end{array}}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A\&B \hookrightarrow (E_1, E_2)}\ \text{T-MERGE}$$

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

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

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

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

Figure 5. Bidirectional type system of λ_i.

$$\boxed{\lfloor A \rfloor = T}$$

$$\begin{aligned}
\lfloor i \rfloor &= i \\
\lfloor x \rfloor &= x \\
\lfloor \lambda x.\, E \rfloor &= \lambda x.\, \lfloor E \rfloor \\
\lfloor e_1\ e_2 \rfloor &= \lfloor e_1 \rfloor \lfloor e_2 \rfloor \\
\lfloor (e_1, e_2) \rfloor &= (\lfloor e_1 \rfloor, \lfloor e_2 \rfloor) \\
\lfloor \texttt{proj}_k e \rfloor &= \texttt{proj}_k \lfloor e \rfloor \quad (k \in 1, 2) \\
\lfloor e_1 ,, e_2 \rfloor &= \lfloor e_1 \rfloor ,, \lfloor e_2 \rfloor \\
\lfloor e : A \rfloor &= \lfloor e \rfloor
\end{aligned}$$

Figure 6. Type annotation erasure.

Uniqueness of type-inference An important property of the bidirectional type-checking in Figure 5 is that, given an expression e, if it is possible to infer a type for e, then e has a unique type.

Theorem 5 (Uniqueness of type-inference)**.** *If* $\Gamma \vdash e_1 \Rightarrow A \hookrightarrow E_1$ *and* $\Gamma \vdash e_2 \Rightarrow A \hookrightarrow E_2$ *then* $e_1 = e_2$.

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

$$\frac{A_1 *_i B \qquad A_2 *_i B}{A_1 \& A_2 *_i B}\ *\&L \qquad \frac{A *_i B_1 \qquad A *_i B_2}{A *_i B_1 \& B_2}\ *\&R$$

$$\frac{A_1 *_i B_1}{A_1 \times A_2 *_i B_1 \times B_2}\ *\pi_1 \qquad \frac{A_2 *_i B_2}{A_1 \times A_2 *_i B_1 \times B_2}\ *\pi_2$$

$$\frac{A_2 *_i B_2}{A_1 \to A_2 *_i B_1 \to B_2}\ * \to \qquad \frac{A *_{ax} B}{A *_i B}\ *\text{AX}$$

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

$$\texttt{Int} *_{ax} A \to B\ *\text{AX}(\mathbb{Z} \to) \qquad \texttt{Int} *_{ax} A \times B\ *\text{AX}(\mathbb{Z} \times)$$

$$A \times B *_{ax} C \to D\ *\text{AX}(\times \to) \qquad \frac{B *_{ax} A}{A *_{ax} B}\ *\text{AXSYM}$$

Figure 7. Algorithmic Disjointness.

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

In contrast, as illustrated in Section 2.3, in the declarative type system some terms may have multiple, incompatible types. Therefore there is no uniqueness of types for the declarative type system, and the same term can have different semantics depending on its type.

4.4 Coherency of Elaboration

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

Theorem 6 (Unique elaboration)**.** *If* $\Gamma \vdash e : A_1 \hookrightarrow E_1$ *and* $\Gamma \vdash e : A_2 \hookrightarrow E_2$*, then* $E_1 \equiv E_2$*. ("≡" means syntactical equality, up to α-equality.)*

Proof. By induction on the first derivation. Note that two cases need special attention: T-SUB and T-APP. In the former, we know that A is unique by Theorem 5 and that C is also unique, by Lemma 5. The latter requires inference of a function type, forcing the use of a type annotation. Again, by Theorem 5 we can use the induction hypotheses. □

5. Algorithmic Disjointness

Section 4 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. The derived set of rules for disjointness is proved to be sound and complete with respect to the definition of disjointness in Section 4.

5.1 Algorithmic Rules

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

Main Judgment. The judgment $A *_i B$ says two types A and B are disjoint. The rules dealing with intersection types ($*\&L$ and $*\&R$) 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 ($* \to$) 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 $A *_i B$ have only taken care of two types with the same language constructs. But how can be the fact that $\texttt{Int}$ 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.

5.2 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 7 (Soundness of algorithmic disjointness)**.** *For any two types A and B, $A *_i B$ implies $A * B$.*

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

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

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

6. Disjoint Intersection Types with $\top$

This section shows how to add a $\top$ type to λ_i. Introducing $\top$ poses some important challenges. Most prominently, the simple definition of disjointness is useless in the presence of $\top$. Since all types now have a common supertype, it is impossible for any two types to satisfy a simple notion of disjointness. To address this problem a notion of $\top$-disjointess is proposed. The definition of $\top$-disjointess depends on a notion of a top-like type. We formalise two different variants of λ_i, based on two different definitions of a top-like type, while discussing their usability and limitations. Both variants retain coherence, and all other key properties of λ_i. Mechanized Coq proofs for both variants are available as part of the supplementary materials for the paper.

$$\text{Types} \quad A, B, C \quad ::= \quad \ldots \mid \top$$

$$\text{Terms} \quad e \quad ::= \quad \ldots \mid \top$$

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

$$\frac{}{A <: \top \hookrightarrow \lambda x.\, ()}\ \textsc{STop}$$

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

$$\frac{}{\Gamma \vdash \top}\ \textsc{WFTop}$$

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

$$\frac{}{\Gamma \vdash \top \Rightarrow \top \hookrightarrow ()}\ \textsc{T-Top}$$

$$\boxed{|A| = T}$$

$$|\top| = ()$$

Figure 8. Extending λ_i with $\top$.

6.1 Introducing $\top$

Introducing the $\top$ type in λ_i is a straightforward process, as shown in Figure 8. Existing types are extended with $\top$ and, correspondingly, we add the canonical inhabitant of type $\top$: the term $\top$. The subtyping relation is extended with STop, declaring that any type is a sub-type of $\top$. The coercion in the target language, is a function that always returns the term $()$, regardless of its argument. We also add $\top$ to the set of well-formed types by extending the well-formedness relation with WFTop. Finally, the typing rule T-Top states that, under type inference, the term $\top$ has type $\top$ and generates the term $()$ in the target language.

6.2 Disjointness

As discussed in Section 2, the definition of simple disjointness is useless when λ_i is extended with $\top$. For these reasons, we differentiate *top-like* types from the rest of the types, so that restrictions may be imposed based on the former. For now, the formal definition of a *top-like* type is omitted, and we informally define it as a type that resembles $\top$ in some way. Having this in mind, $\top$-disjointness is defined as follows:

Definition 4 ($\top$-Disjointness)**.** Given two types A and B we have that:

$$A *_\top B \equiv \neg\rceil A \lceil \wedge \neg\rceil B \lceil \wedge (\forall_C\ (A <: C \wedge B <: C) \to \rceil C \lceil)$$

where $\rceil C \lceil$ means that C is a top-like type.

In other words, given two types A and B:

- A and B cannot be top-like types (i.e. preventing types such as $T\&T$ to be well-formed).

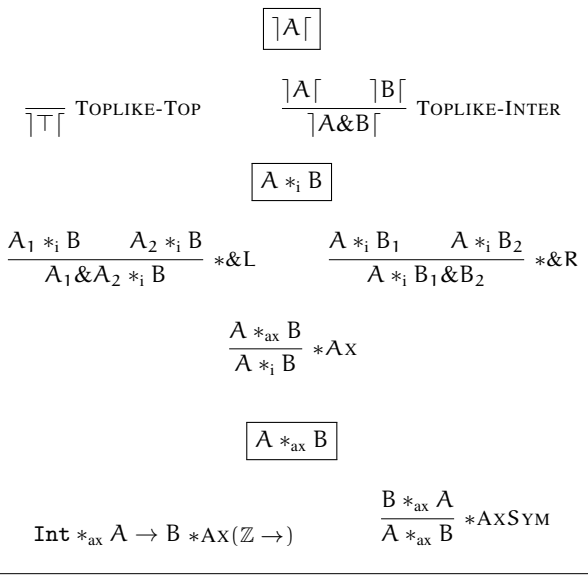

Figure 9. Top-like types and Algorithmic Disjointness.

- If there is any common supertype of A and B, that is not top-like, then intersection of these types is forbidden, as there might be an overlap between them.

BRUNO: I think we need to mention somewhere here that Lemma 4 no longer holds in the presence of $\top$, only a weaker version for non-top-like types. Ultimately this is the key reason why top types make coherence a harder problem in the presence of top.

Next, we will discuss two suitable definitions of top-like and discuss their consequences in a system with $\top$-disjointess.

6.3 A Simple Calculus with $\top$

In the first variant of λ_i with $\top$ the basic idea is to have a definition of top-like types, which captures all syntactically distinct top types: $\top$, $\top\&\top$, $\top\&\top\&\top$. The resulting system has only one *syntactical* $\top$, namely $\top$ itself. Moreover, coherence is preserved.

Top-Like Types A top-like type can be formalised as an unary relation on a type A, denoted as $\lceil A \rceil$, as show in Figure 9. The rule TOPLIKE-TOP states that $\top$ is a top-like type; the rule TOPLIKE-INTER indicates that any intersection composed of just top-like types is also a top-like type.

Algorithmic disjointness rules Similarly to the original system, the definition $\top$-disjointness does not lead to an implementation. Fortunately, the algorithmic disjoitness rules remain the almost same as described in Section **??**. The only significant difference is the absence of the $* \to$ rule. The reason for this is that, in this variant of λ_i, two functions *always* have *non top-like* common supertypes. For example, consider the function types:

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

Although both the domains and co-domains of the functions seem to be unrelated, there are still non top-like common supertypes in the presence of $\top$. For example, $\texttt{Bool\&String} \to \top$ is a common supertype of the previous function types. In general, in this variant of λ_i, any two function types are never disjoint.

Finally, note that this variant of λ_i excludes all types of the form A&B, where A and B are top-like. Thus, the system is left with only one well-formed syntactical $\top$ type.

6.4 An Improved Calculus with $\top$

The former definition of top-like types is, unfortunately, too restrictive. Namely, multiple function types are disallowed within intersection types, which is clearly a limitation from an expressivity angle. As we have seen, both function types $\texttt{String} \to \texttt{Int}$ and $\texttt{String} \to \texttt{String}$ have a new supertype: $\texttt{String} \to \top$. This new supertype is a direct consequence of introducing $\top$ intro our system, and we argue that it is acting also as a top-like type: it represents a function that produces $\top$, no matter what argument it is given. More generally, any type of the form $A_k \to \top$ (with $k \in \mathbb{N}$), can be considered a top-like type. We will extend the definition of top-like type to include types of that form. This will introduce a new ambiguity in our subtyping rules, which will lead us to changing the coercions produced by some of these. Similarly to the simple system, we will show how to re-adjust the algorithmic disjointness rules to match the extended $\top$-disjointness definition.

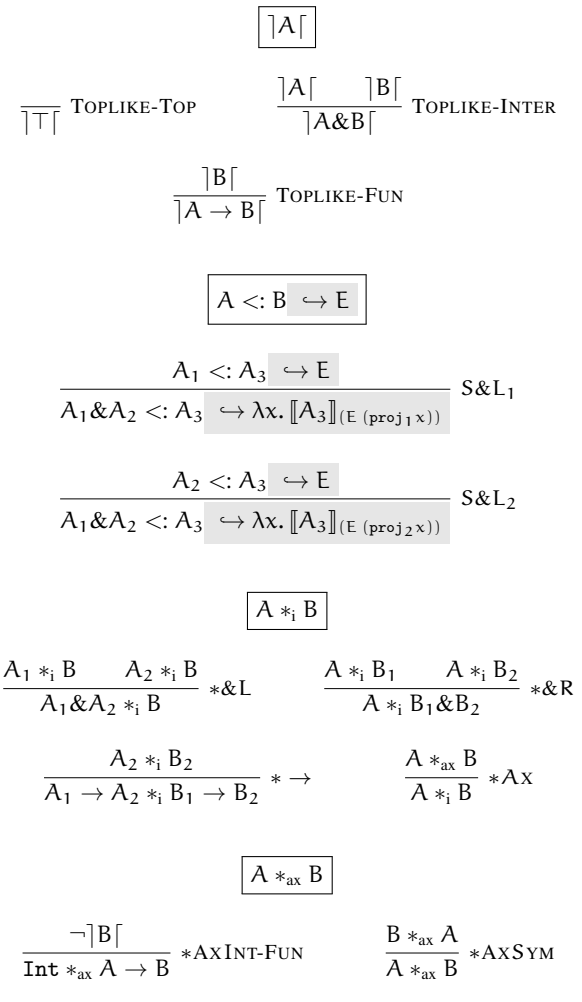

Figure 10. Top-like types, Subtyping (changed rules only) and Algorithmic Disjointness for the improved calculus.

Top-Like Types In relation to the previous definition of top-like, we extend it as follows:

Definition 5 (Top-like types)**.** A type A is (also) a top-like type, if it has the form $A_k \to \top$, where $k \in \mathbb{N}$. That is, any type with arity k can be a top-like type, as long as $\top$ is the result type.

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

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

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

$$[\![A]\!] = \begin{cases} A = \top & \top \\ A = A_1 \to A_2 & \lambda x.\, [\![A_2]\!] \end{cases}$$

Figure 11. Coercion considering intersection of top-like types.

Now, according to our $\top$-disjointess definition, `String` $\to$ `Int` and `String` $\to$ `String` are disjoint and their intersection is a well-formed type. The extended top-like definitions and resulting system are formalised in Figure 10. Note how we just added TOPLIKE-FUN to the top-like relation, by stating that a function is top-like whenever its return type is also top-like. The rest of the changes will be discussed in the following sections.

Coercive Subtyping In this improved calculus, a new problem arises when generating coercions. Introducing functions within intersection types leads to ambiguity between subtype contributors under intersection types. In other words, Lemma ... no longer holds because, under some contexts, S&L$_1$ and S&L$_2$ now overlap. Let us demonstrate this using an example: suppose that we want to build a derivation for `Int` $\to$ `Int&Char` $\to$ `Char` $<:$ (`Int&Char`) $\to \top$ Then, we can either derive it to `Int` $\to$ `Int` $<:$ (`Int&Char`) $\to \top$ (using S&L$_1$), or to `Char` $\to$ `Char` $<:$ (`Int&Char`) $\to \top$ (using S&L$_2$), and thus introducing ambiguity in our system. We could solve this problem at least in two distinct ways:

- Forbid intersection types that include more than one function type; or
- Adjust the subtyping relation to relax its contraints.

Certainly, the first option would be an easy path to take, but the system would not be as expressive as we desired. On the other hand, the second option would require changing the existing rules. Namely, S&L$_1$ and S&L$_2$ should be specialised into four rules, depending whether A_3 is toplike or not. We argue that this would not be very elegant, especially for formalizing the properties of the resulting system. Instead, we observed that, in case A_3 is a top-like type, the same coercion could be generated regardless of the rule that is chosen. Indeed, there is only one way of generating a $()$ term, regardless of the form of the argument(s).

Therefore we opted to modify existing rules to reflect this observation, as shown in Figure 10. Both S&L$_1$ and S&L$_2$ generate coercions that take the source term as argument produce either:

- in case $\rceil A_3 \lceil$: a function with the same arity of A_3, returning $()$;
- otherwise: the same coercion as in the previous systems.

This behaviour is formalised with a function at the type-level, denoted as $[\![A]\!]_C$, as described in Figure 11.

Finally, the reader might notice how the modified rules still overlap, in case $\rceil A_3 \lceil$. In this case both rules can be used interchangeably as they both lead to the same coercion. The rules of the subtyping relation therefore only suffered slight changes while retaining coherence.

Algorithmic disjointness rules Fortunately, the algorithmic disjointness rules are, again, similar to the ones presented in the original system. In relation to the simple system with $\top$ we placed back $* \to$, since we lifted the restriction of intersections with function types. We also had to modify $*\text{AX}(\mathbb{Z} \to)$ to include the premise $\neg\rceil B \lceil$. This is due to the specification of $\top$-disjointness, which requires two types not to be top-like in order for them to be disjoint.

JOAO: add a paragraph mentioning the possibility of using two different definitions of top-like in T-disjointness, so we can get a DisAx-Int-Fun with no premise?

7. Design Space

This section discusses some alternatives in the design-space.

7.1 Disjointness of Functions

Talk about the option of not allowing subtyping of function arguments. This should allow for a more flexible rule for disjointness of functions. Maybe a good option for OO type systems, where methods are invariant with respect to subtyping of arguments. It would allow for static overloading, similar to what is present in conventional OO languages.

$$\frac{A_2 <: B_2}{A_1 \to A_2 <: A_1 \to B_2}\ S \to$$

$$\frac{A_1 *_i B_1}{A_1 \to A_2 *_i B_1 \to B_2}\ * \to$$

not exists E . A -> B <: E / C -> D <: E ->

$\texttt{Int} -> \texttt{Int\&Char} -> \texttt{Int}$ (disjoint according to the spec and algorithmic rules)

Are those 2 functions disjoint?

$f,,g : \texttt{Int} -> \texttt{Int\&Char} -> \texttt{Int}$

Well, two things to consider:

1) what happens if they are applied:

$(f,,g)(3,'c')$

well, a type-annotation will then select one of the functions. So this seems to be ok.

2) what happens if the functions are selected. I have two choices:

f,,g : Int -> Int

f,,g : Char -> Int

$f,,g : \texttt{Char\&Int} -> \texttt{Int}$ (fails because subtyping of functions is invariant).

7.2 Union Types

```
case 3,,'c' of
   Int -> 1
   Char -> 2 : Int
```

Here we have $\texttt{Int\&Char} <: \texttt{Int}|\texttt{Char}$, but this leads to ambiguity. The program can either be 1 or 2.

Possible solution: require atomic constraints in or-rules, similar to the and-rules. Big Problem: subtyping is no longer transitive. Minor problem, type-system is incomplete.

7.3 Parametric Polymorphism?

In principle it should be easy to extend disjointness to parametric polymorphism. brunoShow rules for parametric polymorphism

However, such rules would be quite restrictive. Future work includes how to integrate parametric polymorphism is a more flexible way.

8. Related Work

Coherence. Reynolds invented Forsythe [18] 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 [17] 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 [16] made a comprehensive review of coherence, especially on Curien and Ghelli [8] 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 [12]. 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 [10] 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 [9, 11, 14] 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.* [15] 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 [13] 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 [21]. 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.

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

[8] 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.

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

[10] 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.

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

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

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

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

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