Disjoint Intersection Types

Bruno C. d. S. Oliveira Zhiyuan Shi João Alpuim
The University of Hong Kong
{bruno,zyshi,alpuim}@cs.hku.hk

Abstract

Dunfield showed 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 not *coherent*: different derivations for the same expression can elaborate to expressions that evaluate to different values. 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 coherent and type-safe calculus with a form of *intersection types* and a *merge operator*. Coherence is achieved by ensuring that intersection types are *disjoint* and programs are sufficiently annotated to avoid *type ambiguity*. We propose a definition of disjointness where two types A and B are disjoint only if certain set of types are common supertypes of A and B. We investigate three different variants of λ_i, with three variants of disjointness. In the simplest variant, which does not allow $\top$ types, two types are disjoint if they do not share any common supertypes at all. The other two variants introduce $\top$ types and refine the notion of disjointness to allow two types to be disjoint when the only the set of common supertypes are *top-like*. The difference between the two variants with $\top$ types is on the definition of top-like types, which has an impact on which types are allowed on intersections. We present a type system that prevents intersection types that are not disjoint, as well as an algorithmic specifications to determine whether two types are disjoint for all three variants.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages; F.3.3 [*Studies of Program Constructs*]: Functional constructs

General Terms Design, Languages, Theory

Keywords Intersection Types, Type System

1. Introduction

Intersection types date back to Coppo *et al.* [9] and Pottinger [26] work. Early work was motivated by the applications of intersection types to characterize exactly all strongly normalizing lambda terms. Since then various researchers [6, 12, 15, 17, 22, 24] have started looking at the application of intersection types for designing new and more powerful type systems for programming languages. Since Reynolds' work on the Forsythe [28] programming language, various other languages, including CDuce [4], Scala [20] and Stardust [13] have incorporated some notion of intersection types.

Recently Dunfield [14] showed 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 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 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 source expression can lead to target expressions that evaluate to different values. Dunfield left coherence as an *open problem*. 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 type-safe and coherent core calculus with *intersection types* and a *merge operator*. Coherence is achieved by ensuring that intersection types are *disjoint* and programs are sufficiently annotated to avoid type ambiguity. We propose a definition of disjointness where two types A and B are disjoint when only a certain set of types are common supertypes of A and B. We investigate three different variants of λ_i, with three variants of disjointness. The difference between the variants of disjointness is on what is the set of allowed common supertypes. To avoid type ambiguity, which arises from the undecidability of type assignment of intersection types [30], we use a variant of the calculus that allows type annotations. Without additional type annotations the same term can have multiple types, and consequently different semantics depending on the type. The additional annotations enable a type system based on bidirectional type-checking techniques [16, 25], that ensures that every term has a unique type, eliminating an additional source of ambiguity in the semantics.

In the simplest variant of λ_i, which does not allow $\top$ types, two types A and B are disjoint ($A * B$)[1] if there is no type C such that both A and B are subtypes of 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

[1] The notation $A * B$ is inspired by the *separating conjunction* construct in Reynolds' separation logic [29].

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.

The other two variants of λ_i introduce $\top$ types and refine the notion of disjointness. The main problem of adding $\top$ types into λ_i is that now every two types have at least one common supertype. Therefore, defining disjointness between two types as the non-existence of a common supertype does not work. To account for $\top$ types we propose a notion of $\top$-*disjointness*, where only common supertypes that are *top-like* are allowed for two types A and B. The difference between the two variants with $\top$ types is on the particular definition of top-like types, which has an impact on which types are allowed on intersections. In a simple version, top-like types are the set of types: $\top$, $\top\&\top$, $\top\&\top\&\top$, ... All such types are *proper* top types: that is, they are supertypes of every other type. Unfortunately the simple notion of top-like types has limited expressiveness because it forbids any two function types to be disjoint. The second definition of top-like types is more liberal, and it allows for functional top types (such as $\texttt{Int} \to \top$) to be in the set of top-like types. With this variant of top-like types, the limitations in expressiveness are removed. Both variants with $\top$ types are type-safe and coherent, and both notions of $\top$-disjointness have sound and complete algorithmic versions.

We have mechanized all three variants of λ_i and their meta-theoretical results in the Coq proof assistant. The definitions and proofs use the locally nameless representation with co-finite quantification [3] to represent variables and binders. Except for two auxiliary lemmas used in the lambda cases for the proofs of soundness and completeness of bidirectional type-checking, all proofs are "admit" free. In particular, the main proofs of this paper (coherence and type-preservation) are complete. The two auxiliary theorems that we did not manage to prove are trivially true, but they proved surprisingly tricky to prove using the locally nameless representation with co-finite quantification.

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 for three different variants of disjointness.
- **Formalization of λ_i and Proof of Coherence:** An elaboration semantics of all three variants of λ_i into a simply typed λ-calculus is given. Type-safety and coherence are proved and formalized. All definitions and metatheory are mechanically formalized using the Coq theorem prover.
- **Implementation:** An implementation of λ_i, encodings of several examples presented in the paper, and also the Coq proofs are available at:

  `https://github.com/jalpuim/disjoint-intersection-types`

2. Overview

This section introduces λ_i and its support for intersection types and the merge operator. We also discuss how the notion of disjoint intersection types achieves 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. 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* [6, 14, 27]. As Dunfield [14] has argued a merge operator adds considerable expressiveness to a calculus. The merge operator allows two values to be merged in a single intersection type. 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 [27] (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 [14] 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, $1,,\texttt{True}$ is of type `Int&Bool`. In this case, no matter whether $1,,\texttt{True}$ is used as `Int` or `Bool`, the result of evaluation is always clear. However, with overlapping types, it is not clear 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 [14] 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.

Getting Around Incoherence In a real implementation of Dunfield's 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 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 even though Dunfield has mentioned disjointness as an option to restore coherence, he has not studied the approach further or addressed the issues discussed next.

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 simple 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 simple types, it is less clear 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 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,

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

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 sub-components 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 arising from disjoint intersections, without the additional complications of $\top$. As shown in Section 4, this definition of disjointness is sufficient to ensure coherence in λ_i.

Disjointness of other types Equipped with a formal notion of disjointness, we are now ready to see how disjointness works for other 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 co-domain 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 the definition of simple disjointness for further guidance, and the subtyping rule for functions in λ_i (which is standard [5]):

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

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 $\mathrm{S}{\to}$ 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 look-ups are not ambiguous (and cannot lead to incoherence). If the third intersection type was allowed then the following program:

```
f,,g : (String & Int) → String
```

where `f` has type `Int → String` and `g` has type `String → String`, would be problematic. In this case either `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`?

There is semantic ambiguity in this program, even though it only uses disjoint merges. However a close look reveals a subtly different problem from the previous programs. 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 [31]), types play a fundamental role in determining the result of a program, and the semantics of the language is not completely independent from types. In essence there is some *type ambiguity*, which happens when some expressions can be typed in multiple ways. Depending on which type is chosen for the sub-expressions the program may lead to different results. This phenomenon is common in type-directed mechanisms, and it also affects type-directed mechanisms such as type classes or, more generally, *qualified types* [18].

The typical solution to remove type ambiguity is to add type annotations that choose a particular type for a sub-expression when multiple options exist. Our approach to address this problem is to use bidirectional type-checking techniques [16, 25]. We show that with a fairly simple and unremarkable bidirectional type system, we can guarantee that every sub-expression (possibly with annotations) has a unique type. For example:

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

is a valid λ_i program, which has a unique determined type, and results in `4`. In this case the bidirectional type-system forces users to provide an annotation of the expression being applied. Without the annotation the program would be rejected. With the bidirectional type system, disjointness is indeed sufficient to ensure coherence.

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 can 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 to 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. (not $\rceil A \lceil$) and (not $\rceil B \lceil$)
2. $\forall C$. if $A <: C$ and $B <: C$ then $\rceil C \lceil$

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 $\rceil \cdot \lceil$ 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). A 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 not only $\top\&\top <: \top$ but also $\top <: \top\&\top$ 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 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 reduces the expressiveness 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 to $\top$-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.

Generalizing top-like types In the second variant of λ_i top-like types are defined more liberally and they allow function types whose co-domain is a top type to be considered as a top-like type. For example, the type $\texttt{String} \to \top$ is considered a top-like type. The benefit of allowing this more liberal notion of top-like types is that, similarly to the first variant of λ_i without $\top$, disjoint intersections can have multiple function types.

3. The λ_i Calculus and its Type System

This section presents the syntax, subtyping and type assignment of λ_i: a calculus with intersection types and a merge operator. This calculus is inspired by Dunfield's calculus [14], without considering union types. Moreover, since we are only interested in disjoint intersections, λ_i also has different typing rules for intersections from those in Dunfield's calculus. Sections 4 and 5 will present the more fundamental contributions of this paper by showing other required changes for supporting disjoint intersection types and ensuring coherence. The calculus in this section does not include the

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

Figure 1. λ_i syntax.

$\top$ type, since it brings additional complications. Section 6 presents two variants of λ_i with a $\top$ type, while preserving coherence.

3.1 Syntax

Figure 1 shows the syntax of λ_i. The difference from 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.

Types. Meta-variables 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 `Int`. Note that & has higher precedence than $\to$, meaning that $A\&B \to C$ is equivalent to $(A\&B) \to C$.

Terms. Meta-variables 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 (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 are standard: Γ tracks 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.

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

Note that the notion of ordinary types, which is used in rules S&L$_1$ and S&L$_2$, was introduced by Davies and Pfenning [12] to provide an algorithmic version of subtyping. In our system ordinary types are used for a different purpose as well: they play a fundamental role in ensuring that subtyping produces unique coercions. Section 5 will present a detailed discussion on this. The subtyping relation, is known to be *reflexive* and *transitive* [12].

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

$$A \to B \text{ ordinary} \qquad A \times B \text{ ordinary} \qquad \texttt{Int} \text{ ordinary}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2. Declarative type system of λ_i.

3.3 Declarative Type System

The well-formedness of types and the 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 judgment, 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 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 A can also be assigned the type B.

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

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

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 an 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, even if that issue is fixed, the rule would be counter-intuitive. If A and B are disjoint then, by definition, no expressions 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 safety. 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 and well-formed disjoint intersection types ensures that coercions produced by subtyping relation are unique.
- **No type ambiguity:** the bidirectional type system does not have type-ambiguity, and it infers a unique type for every well-typed expression.

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. We included a unit type $()$ and its only inhabitant $()$, but they will only used in Section 6. The typing rules can be found in our Coq development.

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

Figure 3. Target language syntax.

$\boxed{|A| = T}$

$$|\texttt{Int}| = \texttt{Int}$$
$$|A_1 \to A_2| = |A_1| \to |A_2|$$
$$|A_1 \times A_2| = |A_1| \times |A_2|$$
$$|A_1 \& A_2| = |A_1| \times |A_2|$$

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

$$|\cdot| = \cdot$$
$$|\Gamma, \alpha{:}A| = |\Gamma|, \alpha{:}|A|$$

Figure 4. Type and context translation.

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.

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 a term in the target language and is ensured to have type $|A_1| \to |A_2|$ (by Lemma 1). For example,

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

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.

The rule S$\mathbb{Z}$ generates the identity function as the coercion in the target language. Rule S$\times$ says that one pair is a subtype of another, as long as the first and second component of the former are a subtype of the first and second component of the latter, respectively. The generated coercion extracts both components of the pair, applies the respective coercion, and creates a new pair using both results. In S$\rightarrow$, 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 1 (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. □

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:

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 2 (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 both of the following hold 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 function type, integers and pairs. But in richer systems, ordinary types can also include, for example, record types. 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.

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

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 bidirectional 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 [14]. 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:

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

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, which is closely related to the declarative type-system, presented in Figure 2. The key differences between them 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 $e : A$. The reader might notice that the choice for this type-system does not follow the convention of making introduction forms as checked and elimination forms as synthesized [12, 16]. We use this design because:

1. We aim at maximizing type-inference: annotations becomes less of a burden as there are only two *checked* premises in our rules (excluding T-ANN);
2. Most of the language is simple enough to rely on type inference (i.e. it is decidable);
3. The simplicity of this design allows us to concentrate on its key property, which is the coherence of the type-system.

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

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

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

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

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

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

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

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

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

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

Figure 5. Bidirectional type system of λ_i.

Having this in mind, we do not claim this bidirectional type-system is the optimal choice for the system, and other design choices could be equally justified had our intentions been different.

We may now explain in more detail this elaborating bidirectional type-system. The two elaboration judgments $\Gamma \vdash e \Rightarrow A \hookrightarrow E$ and $\Gamma \vdash e \Leftarrow A \hookrightarrow E$ extend the usual 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 arising from subtyping. The additional coercions are necessary to ensure that the target terms are correctly typed, since the target language lacks subtyping. Note that the elaboration judgments can be modeled as a single relation, where the mode is an additional parameter of the relation. Thus for theorems that need both judgments simple induction is sufficient (mutual induction is not needed).

Type-safety The type-directed elaboration is type-safe. This property is captured by the following two theorems.

Theorem 1 (Type preservation)**.** *We have that:*

- *If* $\Gamma \vdash e \Rightarrow A \hookrightarrow E$, *then* $|\Gamma| \vdash E : |A|$.
- *If* $\Gamma \vdash e \Leftarrow A \hookrightarrow E$, *then* $|\Gamma| \vdash E : |A|$.

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

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

Figure 6. Type annotation erasure.

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 Completeness 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. To relate terms which are typeable in both systems, we use the definition of erasure shown in Figure 6.

Theorem 3 (Soundness of bidirectional type-checking)**.** *We have that:*

- *If* $\Gamma \vdash e \Rightarrow A \hookrightarrow E$, *then* $\Gamma \vdash \lfloor e \rfloor : A$.
- *If* $\Gamma \vdash e \Leftarrow A \hookrightarrow E$, *then* $\Gamma \vdash \lfloor e \rfloor : A$.

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

Theorem 4 (Completeness of bidirectional type-checking)**.** *If* $\Gamma \vdash e : A$, *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. □

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* $\Gamma \vdash e \Rightarrow A_1 \hookrightarrow E_1$ *and* $\Gamma \vdash e \Rightarrow A_2 \hookrightarrow E_2$ *then* $A_1 = A_2$.

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 show the central theorem:

Theorem 6 (Unique elaboration)**.** *We have that:*

- *If* $\Gamma \vdash e \Rightarrow A_1 \hookrightarrow E_1$ *and* $\Gamma \vdash e \Rightarrow A_2 \hookrightarrow E_2$*, then* $E_1 \equiv E_2$.
- *If* $\Gamma \vdash e \Leftarrow A_1 \hookrightarrow E_1$ *and* $\Gamma \vdash e \Leftarrow A_2 \hookrightarrow E_2$*, then* $E_1 \equiv E_2$.

("$\equiv$" 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 T-SUB rule:

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

we need to show not only that E_{sub} is unique (by Lemma 3), but also that A is unique (by Theorem 5). Uniqueness of A is needed to apply the induction hypothesis. For T-APP:

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

we need to show the uniqueness of A_1 using Theorem 5, in order to apply the induction hypothesis. □

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 yet 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, `Int&Bool` $\to$ `String`. Therefore, two function types

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

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

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

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

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

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

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

Figure 7. Algorithmic Disjointness.

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.

Finally, the rules for pairs ($*\pi_1$ and $*\pi_2$) say that either the first components or the second components must be disjoint. This is due to product subtyping being covariant: disjointness of either component is enough to make it impossible to construct a supertype. For instance, take the following two product types:

$$(\texttt{Int}, \texttt{String}) \qquad (\texttt{Bool}, \texttt{String})$$

We can never come up with a common supertype for these because it does not exist a type C which is both a supertype of `Int` and `Bool`.

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 the fact that `Int` and `Int` $\to$ `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 Meta-theory

The following two theorems together say that the algorithmic judgment and the definition of disjointness are equivalent.

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 well-formed types* A *and* B*,* $A * B$ *implies* $A *_i B$.

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

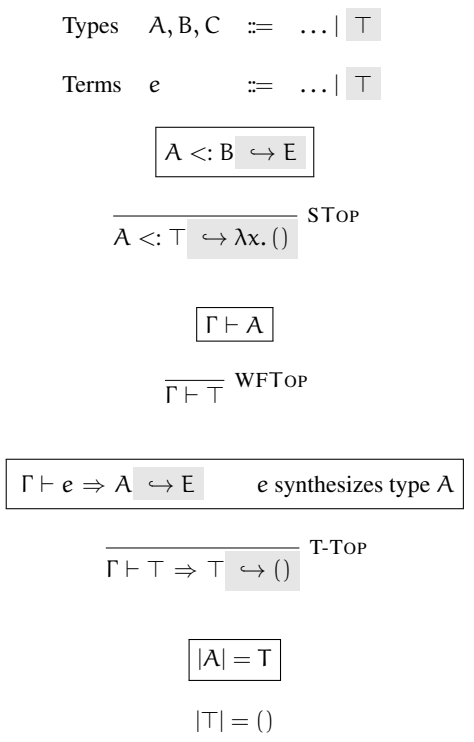

Figure 8. Extending λ_i with $\top$.

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 $\top$-disjointness is proposed. The definition of $\top$-disjointness depends on a notion of a top-like type. We formalize two different variants of λ_i, based on two different definitions of a top-like type, while discussing their expressiveness. Both variants retain coherence, and all other key properties of λ_i. Since the properties are essentially the same as those stated in Sections 4 and 5, we do not restate them here. Also we omit pairs from the two variants of λ_i presented in this section, as they do not impose any important challenge and their introduction is quite straightforward. Mechanized Coq proofs for both variants are available online.

6.1 Introducing $\top$

Introducing the $\top$ type in the λ_i calculus is not difficult. We extended our system following Dunfield's calculus [14], 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 subtype 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:

$$A *_\top B = (\text{not } \rceil A \lceil) \text{ and } (\text{not } \rceil B \lceil) \text{ and } (\forall C.\ \text{if } A <: C \text{ and } B <: C \text{ then } \rceil C \lceil)$$

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

Informally, 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).
- 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.

Another problem arising from the introduction of $\top$ is that Lemma 2 no longer holds. For example, given $A_1 \& A_2 <: \top$ (for any type A_1 and A_2), both A_1 or A_2 can be coerced to $\top$ (because any type is a subtype of $\top$). Thus, in this case, the type $A_1 \& A_2$ has two subtype contributors violating Lemma 2. Nevertheless, it is still possible to prove a weaker result. Namely:

Lemma 4 (Unique subtype contributor (with $\top$))**.** *If $A_1 \& A_2 <: B$, where $A_1 \& A_2$ and B are well-formed types,* ***and B is not top-like****, then it is not possible that the following holds at the same time:*

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

This suggests that we will need to take special care to preserve coherence in the presence of top-like types.

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 *syntactic* $\top$, namely $\top$ itself. Moreover, coherence is preserved.

Top-Like Types A top-like type can be formalized as a unary relation on a type A, denoted as $\rceil A \lceil$, as shown 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 of $\top$-disjointness does not lead to an implementation. Fortunately, the algorithmic disjointness rules, shown in Figure 9, remain the almost same as described in Section 5. The only significant difference is the absence of the $*\!\!\rightarrow$ 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.

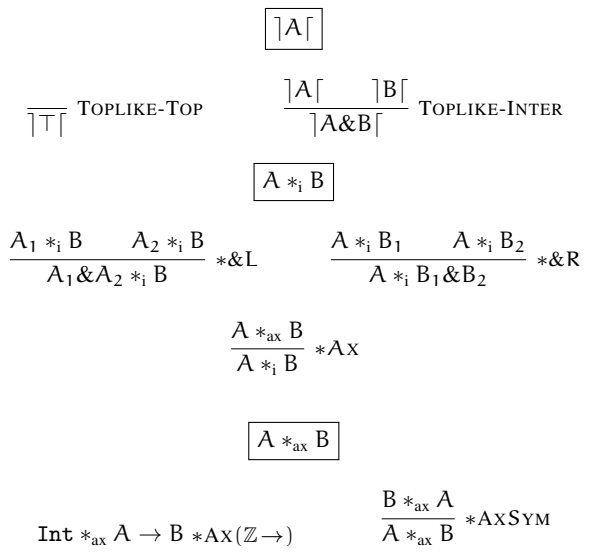

Figure 9. Top-like types and Algorithmic Disjointness.

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 syntactic $\top$ type.

6.4 An Improved Calculus with $\top$

The definition of top-like types in Section 6.3 is, unfortunately, quite restrictive: multiple function types are not allowed within intersection types. This is in contrast with the original λ_i calculus, where multiple function types can co-exist in an intersection. To address this limitation, we generalize the definition of top-like types. The generalization introduces a new ambiguity in our subtyping rules, which requires some changes on the generation of target language coercions.

Top-Like Types The extended top-like definitions and resulting system are formalized 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. Although function types that return a top-like type are not strictly speaking top-types, they can be viewed as *pre-top-types*: applying a value of this type results in a value of type $\top$, regardless of the application's argument(s). In general, any type of the form $A_k \to \top$ (with $k \in \mathbb{N}$), is considered a top-like type in the new variant of λ_i. The consequence of allowing this more liberal notion of top-like types is that now disjoint intersections such as

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

are well-formed: both functions types are not top-like types according to the new definition; and the only common supertypes that they share are top-like types (for example: $\texttt{Bool}\&\texttt{String} \to \top$).

Note that the notion of a *pre-top-type* is not novel amongst intersection type-systems. For example, Coppo et al. [9] refer to function types where the codomain is not $\top$ as *tail-proper* types. However we have not investigated whether the definitions are deeply connected and doing so is left as future work.

Coercive Subtyping The new definition of top-like types introduces a new problem when generating coercions. Introducing functions within intersection types leads to ambiguity between subtype contributors under intersection types. Let us demonstrate this using an example. Suppose that we want to build a derivation for:

$$(\texttt{Int} \to \texttt{Int})\&(\texttt{Char} \to \texttt{Char}) <: (\texttt{Int}\&\texttt{Char}) \to \top$$

There are two possible derivations:

- one using $\texttt{Int} \to \texttt{Int} <: (\texttt{Int}\&\texttt{Char}) \to \top$ (via $S\&L_1$);
- another using $\texttt{Char} \to \texttt{Char} <: (\texttt{Int}\&\texttt{Char}) \to \top$ (via $S\&L_2$).

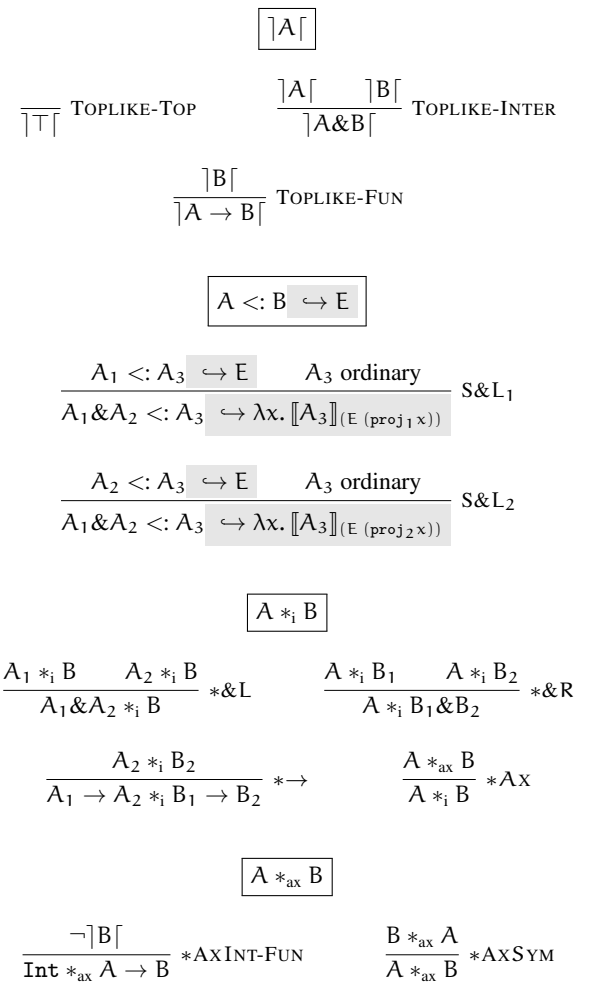

Figure 10. Top-like types, Subtyping (changed rules only) and Algorithmic Disjointness for the improved calculus.

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

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

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

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

Figure 11. Coercion generation considering Top-like types.

Unfortunately the two derivations generate *different* coercions. Thus, without further changes in λ_i, this would be a source of incoherence.

To address this new source of incoherence, we change the way coercions are generated in $S\&L_1$ and $S\&L_2$. The changes are shown in Figure 10. The basic idea is to look at the form of A_3 in both rules:

- when A_3 is top-like ($\lceil A_3 \rceil$), the coercion is a function with the same arity of A_3, returning $()$;
- when A_3 is not top-like: the same coercion (as in the previous systems) is generated.

This behavior is formalized with a meta-function, denoted as $[\![A]\!]_C$, and described in Figure 11.

Finally, the reader may notice how the modified rules still overlap when $\lceil A_3 \rceil$. However, in this case, both rules can be used interchangeably as they both lead to the *same coercion*. With this change in the subtyping rules we are able to retain coherence.

Algorithmic disjointness rules 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 $*\!\!\rightarrow$, since we lifted the restriction of intersections with function types. We also had to modify $*Ax(\mathbb{Z}\rightarrow)$ to include the premise $\neg\lceil B \rceil$. 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.

7. Related Work

Merge Operator. Reynolds invented Forsythe [28] in the 1980s. Forsythe has a merge operator operator p_1, p_2 and a coherent semantics. The result was proved formally by Reynolds [27] in a lambda calculus with intersection types and a merge operator. 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. With those four rules, a merge can only be constructed when the second component of the merge is either a function or a (one-field) record. The set of rules is restrictive and it forbids, for instance, the merge of two functions. In λ_i a merge can contain, for example two primitive values (which is disallowed in Forsythe). Moreover, except for the variant presented in Section 6.3, multiple functions can co-exist in a merge as long as they are provably disjoint. The treatment of disjointness of functions is particularly challenging, specially in combination with a $\top$ type, and supporting multiple functions (as well as other types) in a merge is a significant innovation over Reynolds' approach.

Castagna et al. [6] proposed $\lambda\&$ to study the overloading problem for functions. Their calculus contains a special merge operator that works only for functions. The calculus is coherent. Similarly to us they impose well-formedness conditions on the formation of a (functional) merge. However, their well-formedness conditions cannot be ported to a system with arbitrary intersections like λ_i, since those conditions assume function types only. They also show how to encode records, but it seems that encoding arbitrary merges and intersections is not possible. For the special case of functional merges, the conditions that are used in $\lambda\&$ are incomparable in expressive power to those in λ_i. That is, $\lambda\&$ accepts some functional merges that λ_i rejects, and λ_i accepts some functional merges that $\lambda\&$ rejects. For example, the functional merge

$$(\texttt{Int} \rightarrow \texttt{Char})\&(\texttt{Int} \rightarrow \texttt{Bool})$$

is accepted in λ_i but not in $\lambda\&$. One reason why λ_i also rejects some functional intersections, which are accepted in $\lambda\&$, seems to be related to the presence of arbitrary merges. As discussed in Section 5, the combination of contravariance and the presence of arbitrary merges means that we can always find a common supertype of two functions that have non-disjoint co-domains. In $\lambda\&$ the non-existence of arbitrary merges means that it is harder to find common supertypes of functions, allowing for a more liberal notion of coherent functional merges.

Our work is largely inspired by Dunfield [14], and throughout the paper we have already made extensive comparisons with his work. 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 coherence. Dunfield also mechanically formalized his proofs, using the Twelf proof assistant [21]. However he did not prove any results about coherence, so his meta-theoretical results were not immediately useful to us. Since we were also not familiar with Twelf, we decided to start a new formalization in Coq (which we are familiar with), while proving many new results related to coherence.

Pierce [23] made a comprehensive review of coherence, especially on Curien and Ghelli [10] 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.

Coherence without Merge. Recently, Castagna *et al.* [7] studied a 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 [8] added intersection types to System F_ω and used a new calculus, $F^\omega_\wedge$, to model multiple inheritance. In their system, types include the construct of intersection of types of the same kind K. Davies and Pfenning [12] studied the interactions between intersection types and effects in call-by-value languages. And they proposed a "value restriction" for intersection types, similar to value restriction on parametric polymorphism. We borrowed the notion of ordinary types from Davies and Pfenning. Ordinary types play a fundamental role in ensuring coherence in λ_i. In contrast to λ_i, 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 those languages there is no explicit introduction construct like our merge operator.

Other Type Systems with Intersection Types. Refinement intersection [11, 13, 17] is the more conservative approach of adopting intersection types. Refinement intersections usually have a restriction on the formation of an intersection type A&B. In refinement intersections A&B is a well-formed type if A and B are refinements of the same simple (unrefined) type. However this is a different restriction from disjointness. Refinement intersection 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.

Coherence in other Type-Directed Mechanisms Other type-directed mechanisms such as type classes [31] and, more generally, qualified types [19] also require special care to ensure coherence. For example, in Haskell, a well-known example of ambiguity [18] that could lead to incoherence is:

```
f :: String -> String
f s = show (read s)
```

Here the functions `read` and `show` have, respectively the types `Read a => String -> a` and `Show a => a -> String`. The constraints `Read a` and `Show a` represent requirements that the compiler should implicitly infer. That is the compiler should find a suitable implementation of the read and show functions for the particular type `a`. The problem is that there is not enough type information to determine what the type `a` should be. An arbitrary choice of `a` could lead to different code being inferred depending on the particular choice of `a`. In this case the behavior of the program would be dependent on the particular choice made by the compiler. As a result Haskell compilers, reject programs like the above. Jones [18] provided a formal treatment of coherence for qualified types and type classes, designing suitable restrictions that ensure that incoherent programs are not accepted. In recent versions of Haskell compilers, it is possible to use type annotations to remove type ambiguity:

```
f :: String -> String
f s = (show :: Int -> String) (read s)
```

Here the type annotation instantiates `a` to `Int`, removing the ambiguity and allowing the program to be accepted. In λ_i we also use annotations to remove type ambiguity. The bidirectional type system ensures that the additional annotations required in terms are sufficient to remove any type ambiguity in λ_i programs.

8. Conclusion and Future Work

This paper described λ_i: a coherent and type-safe core calculus that combines intersection types and a merge operator. We investigated three different variants of λ_i: two variants with a $\top$ type; and another one without. To ensure coherence the type system accepts only disjoint intersections. For each variant of λ_i there is a different definition of disjointness. Nevertheless all definitions 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.

For the future, 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, and that is not a top-like type. Another interesting challenge is to address the combination between disjoint intersection types and polymorphism. A naive combination does not seem to be difficult. Since an expression with a polymorphic type can be instantiated to *any* type, a simple option is simply to forbid polymorphic variables in intersections. However this has limited expressiveness, and would prevent many useful programs. More thought is needed to achieve more expressiveness. Finally more work is needed to illustrate the practical applicability of this calculus. One direction that we started exploring is the design of object-oriented languages with sophisticated multiple inheritance mechanisms. The idea is to use the merge operator, combined with fixpoints, to encode multiple inheritance. The role of disjointness is then to remove the ambiguity caused by multiple inheritance.

Acknowledgments

We would like to thank the ICFP reviewers for their helpful comments. This work has been sponsored by the Hong Kong Research Grant Council Early Career Scheme project number 27200514.

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] B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In G. C. Necula and P. Wadler, editors, *Proceeding of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 3–15. ACM, 2008.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, 2003.

[5] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[6] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, 1992.

[7] 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.

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

[9] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2–6):45–58, 1981.

[10] 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.

[11] R. Davies. *Practical refinement-type checking*. PhD thesis, Carnegie Mellon University, 2005.

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

[13] J. Dunfield. Refined typechecking with Stardust. In A. Stump and H. Xi, editors, *Programming Languages meets Program Verification (PLPV '07)*, Freiburg, Germany, Oct. 2007.

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

[15] J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In A. Gordon, editor, *Foundations of Software Science and Computation Structures (FOSSACS '03)*, pages 250–266, Warsaw, Poland, Apr. 2003. Springer-Verlag LNCS 2620.

[16] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, 2004.

[17] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, Toronto, Ontario, June 1991. ACM Press.

[18] M. P. Jones. Coherence for Qualified Types. Technical Report YALEU/DCS/RR-989, Yale University, 1993.

[19] M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, England, 1994.

[20] M. Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[21] F. Pfenning and C. Schürmann. System description: Twelf — A metalogical framework for deductive systems. In H. Ganzinger, editor,

Proceedings of the International Conference on Automated Deduction, pages 202–206, 1999.

[22] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[23] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991.

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

[25] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1), Jan. 2000.

[26] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577. Academic Press, London, 1980.

[27] 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.

[28] J. C. Reynolds. *Design of the Programming Language Forsythe*, pages 173–233. Birkhäuser Boston, Boston, MA, 1997.

[29] 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.

[30] S. van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Comput. Sci.*, 99, 1992.

[31] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, 1989.