A Calculus with Recursive Types, Record Concatenation and Subtyping

No Author Given

Abstract. Calculi with subtyping, a form of record concatenation and recursive types are useful to model objects with multiple inheritance. Surprisingly, almost no existing calculi supports the three features together, partly because the combination of subtyping and record concatenation is already known to be troublesome. Recently, a line of work on *disjoint intersection types* with a *merge operator* has emerged as a new approach to deal with the interaction between subtyping and record concatenation. However, the addition of recursive types has not been studied.
In this paper we present a calculus that combines *iso-recursive types* with disjoint intersection types and a merge operator. The merge operator generalizes symmetric record concatenation, and the calculus supports subtyping as well as recursive types. We build on recent developments on the theory of iso-recursive subtyping using the so-called *nominal unfolding* rules to add iso-recursive types to a calculus with disjoint intersection types and a merge operator. The main challenge lies in the disjointness definition with iso-recursive subtyping. We show the type soundness of the calculus, decidability of subtyping, as well as the soundness and completeness of our disjointness definition. All the proofs are mechanized in the Coq theorem prover.

1 Introduction

Record calculi with a concatenation operator have attracted the attention of researchers due to their ability to give the semantics of object-oriented languages with multiple inheritance [19,14,15]. The foundational work by Cook and Palsberg [19], and Cardelli [14] work on the semantics of the Obliq language are prime examples of the usefulness of untyped record calculi with record concatenation to model the semantics of OOP with inheritance.

Unfortunately, typed calculi with record concatenation and subtyping have proven to be quite challenging to model. An important problem, identified by Cardelli and Mitchell [15], is that subtyping can hide static type information that is needed to correctly model (common forms of) record concatenation. Cardelli and Mitchell illustrate the problem with a simple example:

```
let f2 (r:{x:Int}) (s:{y:Bool}) : {x:Int} & {y:Bool} = r,,s
in f2 ({x=3, y=4}) ({y=true, x=false})
```

Here `f2` is a function that takes two records (`r` and `s`) as arguments, and returns a new record that concatenates the two records (`r ,, s`). For the return type of `f2` we use record type concatenation (here denoted as `R & S`). Because of

subtyping it is possible to invoke `f2` with records that have *more* fields than the fields expected by the static types of the arguments of `f2`. For instance, while the static type of the first argument of `f2` is `{x : Int}`, the record that is actually provided at the application (`{x = 3, y = 4}`) also has an extra field `y`.

The program above is fine from a typing point of view, but what should the program evaluate to? There are a few common options for the semantics of record concatenation. Record concatenation can be *symmetric*, only allowing the concatenation of records without conflicts; or it can be *asymmetric*, implementing an overriding semantics where, in case of conflicts, fields on the left (or the right) record are given preference. Choosing asymmetric concatenation does not work. For instance, with left-biased concatenation, the example above would evaluate to `{x = 3, y = 4}`, which has the *wrong type*! Therefore, Cardelli and Mitchell [15] state that:

> *we should now feel compelled to define R & S only when R and S are* **disjoint***: that is when any field present in an element of R is absent from every element of S, and vice versa.*

hinting for an approach with symmetric concatenation, based on disjointness. But a naive symmetric concatenation operation would result in a record `{x = 3, y = 4, y = true, x = false}` with conflicts, which should not be allowed! Thus such a naive form of symmetric concatenation does not work either.

Recent work on calculi with *disjoint intersection types* [33] and a *merge operator* offer a solution to the Cardelli and Mitchell's problem for concatenation. For instance, the λ_i calculus [33] adopts disjointness and restricts subsumption to address the challenges of *symmetric* concatenation/merge. Most importantly, λ_i has a type-directed semantics to ensure proper information hiding and the preservation of the expected modular type invariants. The application of the `f2` function in λ_i results in `{x = 3, y = true}`, which has *no conflicts* and is of the *right type*. Types are used at runtime to ensure that fields hidden by subtyping are dropped from the record. This is enforced, for example, during beta-reduction, which uses the type of the argument to filter any hidden fields/-values from records/merges. Thus, before substitution, the first argument of `f2`, for instance, is first filtered using the type `{x : Int}`. The actual record that is substituted in the body of `f2` is `{x = 3}` (and not `{x = 3, y = 4}`).

An important limitation of existing calculi with disjoint intersection types is that they lack recursive types. For the typed model of objects, supporting recursive types is important, since many object encodings require recursive types [10]. Without recursive types *binary methods* [9] and other types of methods, that refer to the current object type cannot be easily modelled. For example, it is hard to support an equality method in an object.

This paper studies subtyping relations combining *iso-recursive subtyping* with disjoint intersection types and a merge operator. Our calculus λ_i^μ, extends λ_i with recursive types. With λ_i^μ we can use a standard encoding of objects using recursive types [10,11,19,18] in λ_i^μ to model objects with recursive types. For instance, we can define an interface for arithmetic expressions Exp using a

recursive type:

$$\mathsf{Exp} := \mu\ \mathsf{Exp}.\ \{\mathsf{eval} : \mathsf{nat},\ \mathsf{dbl} : \mathsf{Exp},\ \mathsf{eq} : \mathsf{Exp} \to \mathsf{bool}\}$$

In Exp there are 3 methods: an evaluation method that returns the value of evaluating the expression; a dbl method that doubles all the natural numbers in (the AST of) an expression; and an equality method that compares the expression with another expression. In λ_i it is only possible to express the type of eval. However, in λ_i^μ we can also express dbl and eq. Importantly, in λ_i^μ the record type $\{\mathsf{eval} : \mathsf{nat},\ \mathsf{dbl} : \mathsf{Exp},\ \mathsf{eq} : \mathsf{Exp} \to \mathsf{bool}\}$ is syntactic sugar for *intersections of single field records* [40,22]. In other words, to define the type Exp we need both intersection types and recursive types.

To add iso-recursive subtyping to λ_i^μ, we employ a recent formulation of iso-recursive subtyping based on the so-called *nominal unfolding* rules [46]. The nominal unfolding rules have equivalent expressive power to the iso-recursive Amber rules [12], but they are easier to work with formally. We prove various important properties for λ_i^μ, including *transitivity* of algorithmic subtyping, *decidability* as well as the *unfolding lemma*. A key technical challenge is how to define disjointness for iso-recursive types, which turns out to be non-trivial. By employing the notion of a *lower common supertype*, we show that it is possible to obtain a sound and complete formulation of algorithmic disjointness. All the calculi and lemmas presented in this paper have been mechanically formalized in the Coq theorem prover [42]. In summary, the contributions of this paper are:

- **Iso-recursive subtyping with intersection types:** We show the applicability of nominal unfoldings to a subtyping relation that includes intersection types. The subtyping relation is *transitive*, *decidable* and supports the *unfolding lemma*.
- **The λ_i^μ calculus,** which adds iso-recursive types to an existing calculus with record types, disjoint intersection types and a merge/concatenation operator.
- **Algorithmic disjointness for iso-recursive types.** The algorithmic formulation of disjointness for iso-recursive types is non-trivial. We introduce an approach based on *lower common supertypes*, enabling a *sound* and *complete* algorithm for disjointness.
- **Mechanical formalization:** Finally, we provide a mechanical formalization and proofs for all the calculi and proofs in the Coq theorem prover [42]. The proofs are available in the supplementary material of this submission.

2 Overview

2.1 Background: Disjoint Intersection Types

λ_i and other calculi with disjoint intersection types [33,2,6] have been shown to provide flexible forms of *dynamic* multiple inheritance [5,45]. Moreover, they enable a highly modular and compositional programming style that addresses

the Expression Problem [43] naturally. For space reasons, here we only illustrate briefly the ability of such calculi to model *first-class traits* and a very dynamic form of inheritance [5]:

```
addId(super:Trait[Person], idNumber:Int):Trait[Student]=
  trait inherits super => { def id : Int = idNumber }
```

In this code, written in the SEDEL language [5], there are two noteworthy points. Firstly, unlike statically typed mainstream OOP languages, traits (which are similar to OOP classes) are first class. They can be passed as arguments (such as `super`), or returned as a result as above. Secondly, the code uses a highly dynamic form of inheritance. The trait that is inherited (`super`) is a parameter of the function. In contrast, in languages like Java, for `class A extends B`, the class `B` must be statically known. We refer the interested reader to the work by [5,45] for a much more extensive discussion on the applications of calculi with disjoint intersection types, as well as how to encode source language features, such as first-class traits.

2.2 λ_i^μ: Adding recursive types to λ_i

An important limitation of existing calculi with disjoint intersection types is that they lack recursive types, preventing *binary methods* [9] and other types of methods. As we have discussed in Section 1, without recursive types we cannot write object interfaces such as:

$$\mathsf{Exp} := \mu\ \mathsf{Exp}.\ \{\mathsf{eval} : \mathsf{nat},\ \mathsf{dbl} : \mathsf{Exp},\ \mathsf{eq} : \mathsf{Exp} \to \mathsf{bool}\}$$

where some method signatures refer to the type being defined. In λ_i^μ we add recursive types, therefore it becomes possible to define the object interface Exp. Using a standard object encoding based on records and recursive types [18,10], we can then model objects. To implement Exp we first need a few auxiliary functions ($\mathsf{eval}' : \mathsf{Exp} \to \mathsf{nat}$, $\mathsf{dbl}' : \mathsf{Exp} \to \mathsf{Exp}$ and $\mathsf{eq}' : \mathsf{Exp} \to \mathsf{Exp} \to \mathsf{bool}$) that unfold the recursive type[1]. Then we define two recursive functions $\mathsf{lit} : \mathsf{nat} \to \mathsf{Exp}$ and $\mathsf{add} : \mathsf{Exp} \to \mathsf{Exp} \to \mathsf{Exp}$:

$$\begin{array}{lll}
\mathsf{eval}'\ \mathsf{e} & = (\mathsf{unfold}\ [\mathsf{Exp}]\ \mathsf{e}).\mathsf{eval} & \\
\mathsf{dbl}'\ \mathsf{e} & = (\mathsf{unfold}\ [\mathsf{Exp}]\ \mathsf{e}).\mathsf{dbl} & \\
\mathsf{eq}'\ \mathsf{e}_1\ \mathsf{e}_2 & = (\mathsf{unfold}\ [\mathsf{Exp}]\ \mathsf{e}_1).\mathsf{eq}\ \mathsf{e}_2 & \\
\mathsf{lit}\ \mathsf{n} & = \mathsf{fold}[\mathsf{Exp}]\{\mathsf{eval} = \mathsf{n},\ \mathsf{dbl} = \mathsf{lit}(\mathsf{n} * 2), & \\
 & \qquad\qquad \mathsf{eq} = \lambda \mathsf{e}'.\ (\mathsf{eval}'\ \mathsf{e}' == \mathsf{n}) : \mathsf{Exp} \to \mathsf{bool}\} & \\
\mathsf{add}\ \mathsf{e}_1\ \mathsf{e}_2 & = \mathsf{fold}[\mathsf{Exp}]\{\mathsf{eval} = \mathsf{eval}'\ \mathsf{e}_1 + \mathsf{eval}'\ \mathsf{e}_2,\ \mathsf{dbl} = \mathsf{add}\ (\mathsf{dbl}'\ \mathsf{e}_1)\ (\mathsf{dbl}'\ \mathsf{e}_2), & \\
 & \qquad\qquad \mathsf{eq} = \lambda \mathsf{e}'.\ (\mathsf{eval}'\ \mathsf{e}' == \mathsf{eval}'\ \mathsf{e}_1 + \mathsf{eval}'\ \mathsf{e}_2) : \mathsf{Exp} \to \mathsf{bool}\} &
\end{array}$$

In this example the functions lit and add act as encodings of classes or traits. The function lit is basic: it stores the literal, a double function and equality functions.

[1] We assume the presence of recursive functions, and that records are lazy in the example.

In add, operations such as eval′ have to be called for subexpressions. To check if $2 * 7 = 2 * (3 + 4)$, we can define $\mathsf{e}_1 : \mathsf{Exp} = \mathsf{lit}\ 7$ and $\mathsf{e}_2 : \mathsf{Exp} = \mathsf{add}\ (\mathsf{lit}\ 3)\ (\mathsf{lit}\ 4)$. Then, we check if $\mathsf{eq}'\ (\mathsf{dbl}'\ \mathsf{e}_1)\ (\mathsf{dbl}'\ \mathsf{e}_2)$ is satisfied.

2.3 Disjointness for Recursive Types

The disjointness restriction is an essential feature in calculi with disjoint intersection types. Such restriction ensures that certain merges of values, that could lead to ambiguity, are forbidden. For instance, in the example above merges are used to encode records. A record $\{x = 1, y = true\}$ is encoded as the merge of two single field records $\{x = 1\},,\{y = true\}$. Here the operator ,, is the merge operator [40,22], which can be viewed as a generalization of record concatenation. Ambiguity can arise with the merge operator if the two values in the merge overlap. For instance, with records, we would like to forbid $r = \{x = 1, x = 2\}$ (a record with two fields with the same name and type), since $r.x$ would be ambiguous. With the merge operator we can merge not only records, but also arbitrary values. Thus, we need to forbid merges such as $1,,2$ which provides two values of type Int. The disjointness restriction is employed when type-checking merges to ensure that the types being merged do not overlap. A standard specification of disjointness [33,29] is:

Definition 1 (Specification of disjointness). $\Gamma \vdash A *_s B \equiv \forall C, (\Gamma \vdash A \leq C \wedge \Gamma \vdash B \leq C) \Rightarrow \rceil C \lceil$

The intuition is that two types are disjoint when all their supertypes are (isomorphic to) $\top$. The notation $\rceil \cdot \lceil$ represents toplike types, which are both supertypes and subtypes of $\top$. In essence ambiguity arises from upcasts on values. For instance if we cast $1,,2$ under type Int there can be two possible results. Disjointness prevents merges with values having common supertypes (with the exception of $\top$). Therefore, when such disjoint merges are upcast we can ensure that only 1 value will be extracted for any given (non top-like) type.

One of the challenges in the design of λ_i^μ is to find an algorithmic rule to check whether two recursive types are disjoint and prove that it is complete with respect to the specification. As part of the completeness proof we must be able to find common supertypes of two types, but this is non-trivial for recursive types due to contravariance. For example, assume that we have two recursive types $\mu\alpha.\ ((\mathsf{nat} \to \alpha) \to \mathsf{nat})$ and $\mu\alpha.\ ((\top \to \alpha) \to \mathsf{nat})$, then $\mu\alpha.\ ((\mathsf{nat} \to \alpha)\&(\top \to \alpha) \to \mathsf{nat})$ is not a valid common supertype because α is contravariant. In contrast, for covariant recursive types and non-recursive types, finding a common supertype is simpler. For instance, for the recursive types $\mu\alpha.\ (\mathrm{String} \to \alpha)$ and $\mu\alpha.\ (\mathsf{nat} \to \alpha)$, the intersection of the two inputs types of the function in the recursive type gives us a common supertype $\mu\alpha.\ (\mathrm{String}\&\mathsf{nat} \to \alpha)$.

In Section 4, we will show the disjoint rules for recursive types are quite simple: we only need to check if their one-time finite unfoldings are disjoint or not. Furthermore, we can address the challenge of finding supertypes using a

lower common supertype definition, which gives a common supertype even for contravariant recursive types. This plays a crucial role in the completeness proof for disjointness.

3 Static Semantics of λ_i^μ

This section presents the static semantics of λ_i^μ, covering syntax, subtyping, disjointness and typing rules. Our subtyping relation supports intersection types and iso-recursive types using the nominal unfolding rule [46]. Among others, we prove *transitivity* of subtyping, the *unfolding lemma* and *decidability* of subtyping. We note that this subtyping relation is quite general and, while it is used by our specific application in Section 4, it can be easily adapted to many other calculi with recursive types and intersection types.

3.1 Syntax and Subtyping

Syntax The syntax of our calculus is:

$$
\begin{array}{lrl}
\text{Types} & A, B ::= & \mathsf{nat} \mid \top \mid \bot \mid A_1 \to A_2 \mid \{\alpha : A\} \mid \alpha \mid \mu\alpha.\ A \mid A_1 \& A_2 \\
\text{Expressions} & e ::= & i \mid \top \mid x \mid \lambda x.\ e : A \to B \mid e_1\ e_2 \mid \mathsf{fix}\ x : A.\ e \mid e : A \\
 & & \mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \mid \{\alpha = e\} \mid e_1, , e_2 \mid e.\alpha \\
\text{Values} & v ::= & i \mid \top \mid \lambda x\ .e : A \to B \mid \mathsf{fold}\ [A]\ v \mid v_1, , v_2 \mid \{\alpha = v\} \\
\text{Contexts} & \Gamma ::= & \cdot \mid \Gamma,\ \alpha \mid \Gamma,\ x : A \\
\text{Modes} & \Leftrightarrow ::= & \Leftarrow \ \mid \ \Rightarrow
\end{array}
$$

Meta-variables A, B range over types. Types are mostly standard and consist of: natural numbers (nat), the top type ($\top$), the bottom type ($\bot$), function types ($A \to B$), type variables (α), and recursive types ($\mu\alpha.\ A$). The most interesting feature is the presence of *labelled types* $\{\alpha : A\}$. Labelled types can be viewed as a simple form of nominal types. They are essentially a pair that contains a name (or type variable) α and a type. We use labelled types in two different ways: 1) we use them with the nominal unfolding rules for iso-recursive subtyping; and 2) we also use them to model records and records types in combination with intersection types and the merge operator.

Expressions, which are denoted as e, include: a top value ($\top$), lambda expressions ($\lambda x.\ e : A \to B$) and fixpoints ($\mathsf{fix}\ x : A.\ e$). Note that for lambda expressions, we annotate both input and output types, since the output types are necessary in a TDOS during reduction, which will be described in Section 4.

Values include a canonical top value ($\top$), lambda expressions ($\lambda x.e : A \to B$), merges of values ($v_1, , v_2$) and record values ($\{\alpha = v\}$). For proving type-safety, the contexts also store the types of variables used in the program. We employ bi-directional type checking in the system, thus $\Leftarrow/\Rightarrow$ represent the checking mode and synthesis mode, respectively.

The syntactic sugar for record types and records is also shown below, illustrates the standard encoding [22,40] in terms of intersection types, labelled types

$\boxed{\Gamma \vdash A}$ *(Well-Formed Type)*

$$\textsc{wft-Nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{nat}} \qquad \textsc{wft-Top}\ \frac{\vdash \Gamma}{\Gamma \vdash \top} \qquad \textsc{wft-Bot}\ \frac{\vdash \Gamma}{\Gamma \vdash \bot} \qquad \textsc{wft-var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{wft-arr}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

$$\textsc{wft-rcd}\ \frac{\Gamma \vdash A}{\Gamma \vdash \{\alpha : A\}} \qquad \textsc{wft-rec}\ \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \mu\alpha.\ A} \qquad \textsc{wft-and}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}$$

Fig. 1: Well-formedness.

and merges.

$$\{\alpha_1 : A_1, \ldots, \alpha_n : A_n\} \equiv \{\alpha_1 : A_1\}\ \&\ \ldots\ \&\ \{\alpha_n : A_n\}$$
$$\{\alpha_1 = e_1, \ldots, \alpha_n = e_n\} \equiv \{\alpha_1 = e_1\}\ ,,\ \ldots\ ,,\ \{\alpha_n = e_n\}$$

Well-Formedness The definition of well-formed types is mostly standard, as Figure 1 shows. An environment is well-formed if all the variables are distinct.

Subtyping The Figure 2 shows the subtyping relation. Rule S-BOT states that any well-formed type A is a supertype of the $\bot$ type. Rule S-VAR is a standard rule for type variables: variable α is a subtype of itself. The rule for function types (rule S-ARROW) and intersection types are standard. Rule S-RCD states that a labelled type is a subtype of another labelled type if the two types are labelled with the same name and $A \leq B$.

Rule S-REC, the nominal unfolding rule, is the most interesting one. In this rule, the body of the recursive type is unfolded twice. However, for the innermost unfolding, the type that we substitute is not the recursive type nor the recursive body directly. Instead, we use a labelled type, where the label has the same name as the recursive variable α, and the type that is labelled is the body of the recursive type. In other words, we substitute the recursive variable by $[\alpha \mapsto \{\alpha : A\}]\ A$. The nominal unfolding rules have been shown by Zhou et al. [46] to have equivalent expressive power to the well-known (iso-recursive) Amber rules [12]. However, the nominal unfolding rules are easier to work with in terms of proofs, which is the reason why we employ them here. In particular, they enable us to prove transitivity, and to have a set of algorithmic rules (without transitivity built in). We refer interested readers to the work by Zhou et al. for the equivalence proofs with respect to the Amber rules and the theory of the nominal unfolding rules.

A toplike type, whose definition is shown as rule S-TOP, is both a supertype and a subtype of $\top$. In calculi with disjoint intersection types, the definition of toplike types plays an important role, since disjointness is defined in terms of toplike types. Allowing a larger set of toplike types enables more types to be disjoint. In particular, the motivation for λ_i to include rule TOP-ARROW in

$\boxed{\rceil A \lceil}$ *(Toplike Type)*

$$\textsc{top-base}\ \frac{}{\rceil \top \lceil} \qquad \textsc{top-and}\ \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \& B \lceil} \qquad \textsc{top-arrow}\ \frac{\rceil B \lceil \qquad \Gamma \vdash A}{\rceil A \to B \lceil} \qquad \textsc{top-rec}\ \frac{\rceil A \lceil}{\rceil \mu\alpha.\ A \lceil} \qquad \textsc{top-rcd}\ \frac{\rceil A \lceil}{\rceil \{\alpha : A\} \lceil}$$

$\boxed{\Gamma \vdash A \leq B}$ *(Subtyping)*

$$\textsc{s-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{nat} \leq \mathbf{nat}} \qquad \textsc{s-top}\ \frac{\vdash \Gamma \qquad \rceil B \lceil}{\Gamma \vdash A \leq B} \qquad \textsc{s-bot}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash \bot \leq A} \qquad \textsc{s-rcd}\ \frac{\Gamma \vdash A \leq B}{\Gamma \vdash \{\alpha : A\} \leq \{\alpha : B\}}$$

$$\textsc{s-arrow}\ \frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{s-and}\ \frac{\Gamma \vdash A \leq B_1 \qquad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \& B_2} \qquad \textsc{s-fvar}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha \leq \alpha}$$

$$\textsc{s-andR}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \& A_2 \leq B} \qquad \textsc{s-andL}\ \frac{\Gamma \vdash A_2 \qquad \Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \& A_2 \leq B}$$

$$\textsc{s-rec}\ \frac{\Gamma, \alpha \vdash [\alpha \mapsto \{\alpha : A\}]\ A \leq [\alpha \mapsto \{\alpha : B\}]\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}$$

Fig. 2: Subtyping rules.

subtyping is to allow certain function types to be disjoint [28,33]. The rule TOP-ARROW itself is inspired from the well-known BCD-subtyping [4] relation, which also states that any function type that returns a top-like type is itself toplike. Without such rule two function types can never be disjoint, disallowing more than one function in a merge (where all expressions must have disjoint types). Similarly, in λ_i^μ the rule TOP-REC enables merges that can contain more than one expression with a recursive type.

Subtyping is reflexive and transitive:

Theorem 1 (Reflexivity). *If* $\vdash \Gamma$ *and* $\Gamma \vdash A$ *then* $\Gamma \vdash A \leq A$.

Theorem 2 (Transitivity). *If* $\Gamma \vdash A \leq B$ *and* $\Gamma \vdash B \leq C$ *then* $\Gamma \vdash A \leq C$.

Furthermore, we have also proved the unfolding lemma, which plays an important role in type preservation:

Lemma 1 (Unfolding Lemma). *If* $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B$ *then* $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B$.

3.2 Decidability

Decidability of subtyping is a significant property, which is often problematic in many subtyping relations [27,37]. Fortunately, under our new iso-recursive subtyping rules with nominal unfoldings, decidability is easy to prove:

Theorem 3 (Decidability of Subtyping). *If* $\vdash \Gamma$, $\Gamma \vdash A$ *and* $\Gamma \vdash B$, *then* $\Gamma \vdash A \leq B$ *is decidable.*

Informally, looking at Figure 2 we can identify two potential complications in deriving an algorithm from the subtyping relation and showing its termination. The first complication comes from the newly added intersection subtyping rules, which makes the relation not completely syntax directed. In particular, there is overlapping between all three intersection rules. However, this problem is well-known from the literature of intersection types. A standard solution, proposed by Davies and Pfenning [21], is to only apply rule S-andL and rule S-andR for $\Gamma \vdash A_1 \& A_2 \leq B$ when B is *ordinary* (i.e. not an intersection type). This removes the overlapping between rule S-and and rules S-andR and S-andL. For the remaining overlapping between rules S-andR and S-andL backtracking is needed. The same approach can be adopted in an implementation of our subtyping relation. The second complication is that the relation is not structurally recursive because of rule S-rec. In rule S-rec the size of the types in the premise can actually grow. However, the key observation here is that the number of recursive binders will reach the peak and decreases after some unfoldings. We employ the same technique by Zhou et. al [46], to prove decidability with a nominal unfolding rule.

3.3 Disjointness

One of the core judgments of λ_i^{μ} is disjointness. The standard disjointness specification [33] states that two types A and B are disjoint if there is all common supertypes of A and B are toplike, as Definition 1 showed. In other words, A and B are disjoint if there are no non-toplike supertypes. While this definition of disjointness is concise, it is not algorithmic. Thus, a challenge in calculi with disjoint intersection types is to find an algorithmic set of rules that is sound and complete to the disjointness specification.

Figure 3 shows an algorithmic formulation of disjointness. Most rules are standard and follow from previous work [33,28,7]. Toplike types are disjoint with other types (rules dis-topL and dis-topR). Intersection types need to check the disjointness of every component (rules dis-andL and dis-andR). Two labelled types are disjoint if they have distinct labels or the types of the label are disjoint (rules dis-RcdRcd and dis-RcdRcdEq). Two different variables are always disjoint (rule dis-VarVar). Rule dis-ArrArr states that, for two function types, we just need to check if their output types are disjoint or not.

The most interesting one is the disjointness of recursive types. Without toplike types, it could be very simple: any two recursive types are disjoint. However, the introduction of toplike types complicates the interaction between any two

$\boxed{\Gamma \vdash A * B}$ *(Disjointness)*

$$\textsc{dis-topL}\ \frac{\rceil B \lceil}{\Gamma \vdash A * B} \qquad \textsc{dis-topR}\ \frac{\rceil A \lceil}{\Gamma \vdash A * B} \qquad \textsc{dis-andL}\ \frac{\Gamma \vdash A_1 * B \quad \Gamma \vdash A_2 * B}{\Gamma \vdash A_1 \& A_2 * B} \qquad \textsc{dis-andR}\ \frac{\Gamma \vdash A * B_1 \quad \Gamma \vdash A * B_2}{\Gamma \vdash A * B_1 \& B_2}$$

$$\textsc{dis-VarVar}\ \frac{\alpha \neq \beta}{\Gamma \vdash \alpha * \beta} \qquad \textsc{dis-ArrArr}\ \frac{\Gamma \vdash A_2 * B_2}{\Gamma \vdash A_1 \to A_2 * B_1 \to B_2} \qquad \textsc{dis-RcdRcd}\ \frac{\alpha \neq \beta}{\Gamma \vdash \{\alpha : A\} * \{\beta : B\}}$$

$$\textsc{dis-RcdRcdEq}\ \frac{\Gamma \vdash A * B}{\Gamma \vdash \{\alpha : A\} * \{\alpha : B\}} \qquad \textsc{dis-RecRec}\ \frac{\Gamma, \alpha \vdash A * B}{\Gamma \vdash \mu\alpha.\ A * \mu\alpha.\ B} \qquad \textsc{dis-axiom}\ \frac{\Gamma \vdash A *_{axiom} B}{\Gamma \vdash A * B}$$

Fig. 3: Disjointness.

recursive types, as we described in Section 2. Nevertheless, rule DIS-RECREC is surprisingly simple: two recursive types are disjoint if their bodies are disjoint. Finally, two types with different type constructors (e.g. record types and recursive types) are disjoint (rule DIS-AXIOM).

Disjointness soundness The soundness lemma showing that our rules satisfy the specification is straightforward:

Lemma 2 (Soundness). *If* $\Gamma \vdash A * B$ *then* $\Gamma \vdash A *_s B$.

3.4 Completeness of Disjointness

The most challenging part of the formalization of λ_i^μ is to show that algorithmic disjointness is complete with respect to the specification. The difficulty is brought by rule DIS-RECREC. If two recursive types $\mu\alpha.\ A$ and $\mu\alpha.\ B$ satisfy the specification, then for any type C, $\Gamma \vdash \mu\alpha.\ A \leq C \wedge \Gamma \vdash \mu\alpha.\ B \leq C$ implies that C is toplike. By rule DIS-RECREC, we want to prove that any type D satisfying $\Gamma, \alpha \vdash A \leq D \wedge \Gamma, \alpha \vdash B \leq D$ implies that D is toplike. Clearly C and D should be related since in one case C is the supertype of two recursive types, and in the other case D is the supertype of the bodies of the two recursive types. However, the relation between C and D is intricate.

Lower common supertype To help relating C and D, we define a new function $\sqcup$, which is shown in Figure 4. The function $\sqcup$ computes a lower supertype of type A and B. A simplification that we employ in our definition is that types of common supertypes in contravariant positions are all $\bot$. Strictly speaking this means that the supertype that we find is not the lowest one in the subtyping lattice. But in our setting this does not matter, because the disjointness of arrow

$$
\begin{array}{rclrcl}
\top \sqcup A & = & \top & A \sqcup \top & = & \top \\
\alpha \sqcup \alpha & = & \alpha & \alpha \sqcup \beta & = & \top \ (\alpha \neq \beta) \\
A \sqcup B\&C & = & A\&B \sqcup A\&C & A\&B \sqcup C & = & A\&C \sqcup B\&C \\
\bot \sqcup \mu\alpha.\ A & = & \mu\alpha.\ (\bot \sqcup A) & \mu\alpha.\ A \sqcup \bot & = & \mu\alpha.\ (A \sqcup \bot) \\
\bot \sqcup \{\alpha : A\} & = & \{\alpha : \bot \sqcup A\} & \{\alpha : A\} \sqcup \bot & = & \{\alpha : A \sqcup \bot\} \\
\bot \sqcup A_1 \to A_2 & = & \bot \to (\bot \sqcup A_2) & A_1 \to A_2 \sqcup \bot & = & \bot \to (A_2 \sqcup \bot) \\
\mu\alpha.\ A \sqcup \mu\alpha.\ B & = & \mu\alpha.\ (A \sqcup B) & \{\alpha : A\} \sqcup \{\beta : B\} & = & \top \ (\alpha \neq \beta) \\
A_1 \to A_2 \sqcup B_1 \to B_2 & = & \bot \to (A_2 \sqcup B_2) & \{\alpha : A\} \sqcup \{\alpha : B\} & = & \{\alpha : A \sqcup B\}
\end{array}
$$

otherwise: $\bot \sqcup A = A,\ A \sqcup \bot = A,\ A \sqcup A = A,\ A \sqcup B = \top,$

Fig. 4: Lower common supertype.

types (see rule DIS-ARRARR) does not account for input types. If the input types did matter for disjointness then we would likely need a dual definition for finding greater common subtypes, making the definition more involved. We can prove some useful properties for $\sqcup$:

Lemma 3 ($\sqcup$ is supertype). *For any A and B, $\Gamma \vdash A \leq A \sqcup B$ and $\Gamma \vdash B \leq A \sqcup B$.*

Lemma 4. *If $\Gamma \vdash A \leq C$ and $\Gamma \vdash B \leq C$ and $A \sqcup B$ is toplike, then C is toplike.*

Lemma 4 is the most important one: $A \sqcup B$ is not the least common supertype of A and B, but if it is toplike then all supertypes of A and B are toplike. With the previous lemmas we can prove the completeness lemma:

Lemma 5 (Completeness). *For types A and B, if $\Gamma \vdash A *_s B$ then $\Gamma \vdash A * B$.*

3.5 Bidirectional Typing

We use bidirectional type checking in λ_i^μ, following λ_i [29]. Bi-directional type-checking is helpful to eliminate a source of ambiguity (and non-determinism) that arises from an unrestricted subsumption rule in conventional type assignment systems in the presence of a concatenation/merge operator (a point which was also noted by Cardelli and Mitchell [15]). The typing rules are shown in Figure 5. There are two standard modes: $\Gamma \vdash e \Rightarrow A$ synthesises the type A of expression e under the context Γ, and $\Gamma \vdash e \Leftarrow A$ checks if expression e has type A under the context Γ.

Many rules are standard. There are two rules for merge expressions, which follow from previous work [29]. Rule TYPING-MERGE employs a disjointness restriction, and only allows two expressions with disjoint types to be merged. The disjointness restriction prevents ambiguity that could arise merging types with common (non-toplike) supertypes. For instance, if $1, , 2$ would be allowed, then in an expression like $(1, , 2) + 3$ we could have two possible results: 4 and 5. The merge of duplicated values such as $1, , 1$ is not harmful, since no ambiguity

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ *(Typing)*

$$\text{TYPING-SUB}\ \frac{\Gamma \vdash e \Rightarrow A \quad \Gamma \vdash A \leq B}{\Gamma \vdash e \Leftarrow B} \qquad \text{TYPING-TOP}\ \frac{\vdash \Gamma}{\Gamma \vdash \top \Rightarrow \top} \qquad \text{TYPING-VAR}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \text{TYPING-APP}\ \frac{\Gamma \vdash e_1 \Rightarrow A_1 \to A_2 \quad \Gamma \vdash e_2 \Leftarrow A_1}{\Gamma \vdash e_1\ e_2 \Rightarrow A_2}$$

$$\text{TYPING-NAT}\ \frac{\vdash \Gamma}{\Gamma \vdash i \Rightarrow \mathbf{nat}} \qquad \text{TYPING-ANNO}\ \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e :\ A \Rightarrow A} \qquad \text{TYPING-PROJ}\ \frac{\Gamma \vdash e \Rightarrow \{\alpha : A\}}{\Gamma \vdash e.\alpha \Rightarrow A}$$

$$\text{TYPING-RCD}\ \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{\alpha = e\} \Rightarrow \{\alpha : A\}} \qquad \text{TYPING-ABS}\ \frac{\Gamma, x : A_1 \vdash e \Leftarrow A_2}{\Gamma \vdash \lambda x.e : A_1 \to A_2 \Rightarrow A_1 \to A_2}$$

$$\text{TYPING-UNFOLD}\ \frac{\Gamma \vdash e \Leftarrow \mu\alpha.\ A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.\ A]\ e \Rightarrow [\alpha \mapsto \mu\alpha.\ A]\ A} \qquad \text{TYPING-FIX}\ \frac{\Gamma, x : A \vdash e \Leftarrow A}{\Gamma \vdash \mathsf{fix}\ x : A.\ e \Rightarrow A}$$

$$\text{TYPING-MERGE}\ \frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma \vdash e_2 \Rightarrow B \quad \Gamma \vdash A * B}{\Gamma \vdash e_1, , e_2 \Rightarrow A \& B} \qquad \text{TYPING-MERGEV}\ \frac{\cdot \vdash v_1 \Rightarrow A \quad \cdot \vdash v_2 \Rightarrow B \quad v_1 \approx_{spec} v_2}{\Gamma \vdash v_1, , v_2 \Rightarrow A \& B}$$

$$\text{TYPING-FOLD}\ \frac{\Gamma \vdash e \Leftarrow [\alpha \mapsto \mu\alpha.\ A]\ A \qquad \Gamma \vdash \mu\alpha.\ A}{\Gamma \vdash \mathsf{fold}\ [\mu\alpha.\ A]\ e \Rightarrow \mu\alpha.\ A}$$

Fig. 5: Typing.

arises in this case, and such values can arise from reduction. Thus, there is also a rule TYPING-MERGEV, which allows merging two consistent values regardless of their types. The consistency relation is:

Definition 2 (Consistency). $v_1 \approx_{spec} v_2 \equiv \forall A, (v_1 \hookrightarrow_A v_1' \wedge v_2 \hookrightarrow_A v_2') \Rightarrow v_1' = v_2'$

In this relation, two values are consistent if for any type A the typed reduction of those two values produces the same result. We introduce the typed reduction $v_1 \hookrightarrow_A v_2$, which reduces the value v_1 to v_2 under the type A in Section 4.1. A key property relating consistency and disjointness is:

Lemma 6 (Consistency of disjoint values). *If* $\vdash v_1 \Rightarrow A$ *and* $\vdash v_1 \Rightarrow B$ *and* $\vdash A *_s B$ *then* $v_1 \approx_{spec} v_2$.

4 Dynamic Semantics of λ_i^{μ}

We now introduce the Type-Directed Operational Semantics (TDOS) for λ_i^{μ}. TDOS, originally proposed by Huang et al. [28,29], is a variant of small-step operational semantics. In TDOS, type annotations are operationally relevant,

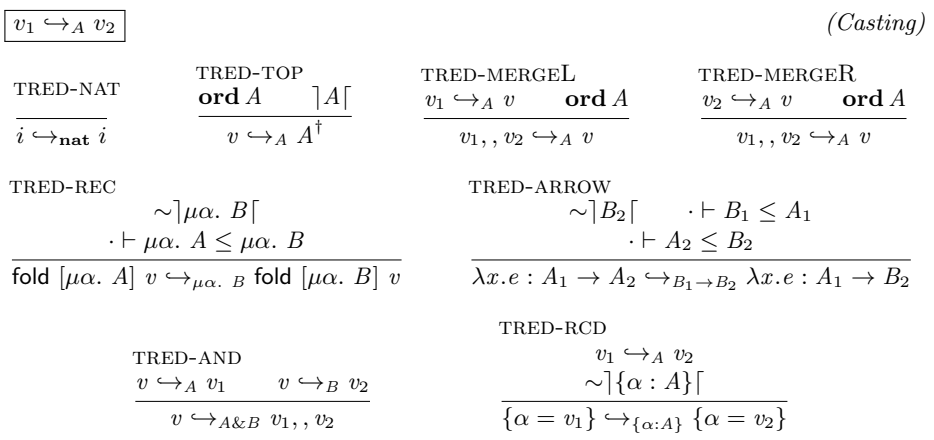

Fig. 6: Casting.

since selecting values from merged values is type-directed. We show that λ_i^μ is deterministic and type-sound.

4.1 A Type-Directed Operational Semantics for λ_i^μ

The defining feature of a TDOS is a relation called casting (originally called typed reduction by Huang et al.). Casting plays an important role: based on the contextual type information, values are further reduced to match the type structure precisely. In many conventional operational semantics a value is the final result in a program, but with TDOS further reduction can happen if the type that is required for the value has a mismatch with the shape of the value. For example, if we have the merge $1, ,' c'$ at type Int then casting will produce 1. However, the same value at type $Int\&Char$ would remain unchanged $(1, ,' c')$.

The rules for casting are shown at the Figure 6. All non-intersection types are ordinary types. Casting $v_1 \hookrightarrow_A v_2$ denotes that the value v_1 is reduced to v_2 under the type A. From the definitions, we can see that the A is the supertype of the principal type of v_1, and v_2 is the value compatible with A. The most special one is rule TRED-TOP: if the type is toplike, then a value will reduce to the corresponding top value, where the $A^\dagger$ is defined as:

$$(A \to B)^\dagger = \lambda x\ .\top : A \to B \quad (\mu\alpha.\ A)^\dagger = \mathsf{fold}\ [\mu\alpha.\ A]\ \top$$
$$\{\alpha : A\}^\dagger = \{\alpha : A^\dagger\} \qquad (A\&B)^\dagger = A^\dagger, , B^\dagger$$

$$\text{otherwise: } A^\dagger = \top$$

4.2 Reduction

The definition of reduction is shown at the Figure 7. Most rules are standard. Casting is used in rule STEP-BETA for adjusting the argument value to the expected type for the input of the function. Casting is also used in rule STEP-ANNOV for annotations. Rules STEP-FLD and STEP-FLDT are for unfold expressions. Finally, there is also a special rule STEP-FLDM for recursive types as well as intersection types.

4.3 Determinism

One of the properties of our semantics is that it is deterministic. That is to say, expressions will always reduce to the same value. Lemma 7 says that if a value can be type-checked, then it reduces to a same value under the type A. Lemma 8 says that if an expression can be type-checked, then it reduces to a unique expression.

Lemma 7 (Determinism of $\hookrightarrow$). *If $\Gamma \vdash v \Rightarrow B$ and $v \hookrightarrow_A v_1$ and $v \hookrightarrow_A v_2$ then $v_1 = v_2$.*

Lemma 8 (Determinism of $\rightsquigarrow$). *If $\Gamma \vdash e \Leftrightarrow A$ and $e \rightsquigarrow e_1$ and $e \rightsquigarrow e_2$ then $e_1 = e_2$.*

4.4 Type safety

We prove type safety following a similar approach to the previous work [29], and by showing progress and preservation theorems. The following lemmas and theorems show that our system is type-safe.

Theorem 4 (Preservation). *If $\vdash e_1 \Leftrightarrow A$ and $e_1 \rightsquigarrow e_2$ then $\vdash e_1 \Leftarrow A$.*

Lemma 9 (Progress of $\hookrightarrow$). *If $\vdash v_1 \Leftarrow A$ then $\exists v_2,\ v_1 \hookrightarrow_A v_2$.*

Theorem 5 (Progress). *If $\vdash e_1 \Leftrightarrow A$ then e_1 is a value or $\exists e_2,\ e_1 \rightsquigarrow e_2$.*

5 Discussion and Related Work

Throughout the paper, we have already reviewed some of the closest related work in detail. In this section, we will discuss other related work.

$\boxed{e_1 \rightsquigarrow e_2}$ *(Reduction)*

STEP-BETA
$$\frac{v_1 \hookrightarrow_{A_1} v_2}{(\lambda x.e : A_1 \to A_2)\ v_1 \rightsquigarrow ([x \mapsto v_2]\ e) :\ A_2}$$

STEP-APPL
$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2}$$

STEP-APPR
$$\frac{e_2 \rightsquigarrow e_2'}{v_1\ e_2 \rightsquigarrow v_1\ e_2'}$$

STEP-PROJ
$$\frac{e \rightsquigarrow e'}{e.\alpha \rightsquigarrow e'.\alpha}$$

STEP-MERGEL
$$\frac{e_1 \rightsquigarrow e_1'}{e_1, , e_2 \rightsquigarrow e_1', , e_2}$$

STEP-MERGER
$$\frac{e_2 \rightsquigarrow e_2'}{v_1, , e_2 \rightsquigarrow v_1, , e_2'}$$

STEP-ANNO
$$\frac{e \rightsquigarrow e'}{e :\ A \rightsquigarrow e' :\ A}$$

STEP-ANNOV
$$\frac{v \hookrightarrow_A v'}{v :\ A \rightsquigarrow v'}$$

STEP-FIX
$$\frac{}{\mathsf{fix}\ x : A.\ e \rightsquigarrow (e\ (\mathsf{fix}\ x : A.\ e)) :\ A}$$

STEP-FOLD
$$\frac{e \rightsquigarrow e'}{\mathsf{fold}\ [A]\ e \rightsquigarrow \mathsf{fold}\ [A]\ e'}$$

STEP-RCD
$$\frac{e \rightsquigarrow e'}{\{\alpha = e\} \rightsquigarrow \{\alpha = e'\}}$$

STEP-UNFOLD
$$\frac{e \rightsquigarrow e'}{\mathsf{unfold}\ [A]\ e \rightsquigarrow \mathsf{unfold}\ [A]\ e'}$$

STEP-FLD
$$\frac{v_1 \hookrightarrow_{[\alpha \mapsto \mu\alpha.\ A]\ A} v_2 \quad \sim\rceil \mu\alpha.\ A \lceil}{\mathsf{unfold}\ [\mu\alpha.\ A]\ (\mathsf{fold}\ [\mu\alpha.\ B]\ v_1) \rightsquigarrow v_2}$$

STEP-FLDM
$$\frac{v_1, , v_2 \hookrightarrow_{\mu\alpha.\ A} v \quad \sim\rceil \mu\alpha.\ A \lceil}{\mathsf{unfold}\ [\mu\alpha.\ A]\ (v_1, , v_2) \rightsquigarrow \mathsf{unfold}\ [\mu\alpha.\ A]\ v}$$

STEP-FLDT
$$\frac{v_1 \hookrightarrow_{[\alpha \mapsto \mu\alpha.\ A]\ A} v_2 \quad \rceil \mu\alpha.\ A \lceil}{\mathsf{unfold}\ [\mu\alpha.\ A]\ v_1 \rightsquigarrow v_2}$$

STEP-PROJRCD
$$\frac{}{\{\alpha = v\}.\alpha \rightsquigarrow v}$$

Fig. 7: Small-step semantics.

Disjoint Intersection Types and Record Calculi with Concatenation. Disjoint intersection types were originally proposed by Oliveira et. al.[33]. Such calculi have intersection types as well as a merge operator [40,22] with a disjointness restriction to ensure the determinism of the language. Follow-up work [2,6,5] provides more advanced features, such as *disjoint polymorphism* and *distributive subtyping* and *first-class traits*, built upon the original work. With all these features together, an alternative paradigm called *Compositional Programming* (CP) is proposed [7,45]. CP allows for a very modular programming style where the Expression Problem [43] can be solved naturally. A limitation of existing calculi with disjoint intersection types is that they do not support recursive types, which are important to encode *binary methods* [9] or, more generally, recursive object types. The λ_i^μ calculus addresses this limitation and shows, for the first time, a calculus with disjoint intersection types and recursive types.

The merge operator generalizes concatenation by allowing values of any types (not just record types) to be merged. As we described in the introduction the interaction between subtyping and record concatenation is quite tricky. Cardelli

and Mitchell observed the problem [15], but did not provide a solution. Instead, they decided to use record extension and restriction operators instead of concatenation. One solution adopted by some calculi [39,38,34,26] is to distinguish between records that can be concatenated, and records that have subtyping. The choice is mutually exclusive: records that can be concatenated cannot have subtyping and vice-versa. Such an approach would prevent Cardelli and Mitchell's `f2` example in the introduction from type-checking. Calculi with disjoint intersection types, including λ_i^{μ}, offer a different solution by adopting a type-directed semantics, which ensures that fields hidden by subtyping are also hidden at runtime. This allows concatenation and subtyping to be used together. As far as we know, no work supports subtyping, record concatenation and recursive subtyping at the same time formally. In Cardelli's $F_{<:_\rho}$ calculus [13] equi-recursive subtyping are assumed to be an extension to record subtyping and record concatenation but no further proof was provided. Palsberg and Zhao's work [34] shows supporting subtyping, record concatenation and recursive types (but no recursive subtyping) together for type inference is NP-complete.

Alternative Models for Typed Objects. There are alternative ways to model objects without having records and record concatenation. Perhaps the most famous one are Abadi and Cardelli's object calculi [1]. In their work, objects are modelled directly. No form of concatenation is provided, but the fields of objects can be updated in object calculi. Besides, they also provide declarative type systems (with a transitivity rule built-in) to support recursive subtyping.

Another alternative approach, which has received a lot of attention recently, are calculi with dependent object types (DOT) [41], which aims to capture the essence of Scala. Many variants of DOT include intersection types, a form of recursive types and subtyping. However, there are no records and record concatenation, since objects are directly modelled rather than being encoded via records. The subtyping rules for intersection types are similar to ours. Moreover, the rules for recursive types employed in some variants of DOT [41,44,25] are mostly structural and employ an inductive definition of subtyping. The key subtyping rules in the DOT variant by [41] are shown next:

$$\frac{\Gamma, z : T_1^z \vdash T_1^z <: T_2^z}{\Gamma \vdash \mu z.\ T_1^z <: \mu z.\ T_2^z}\ \text{BindX} \qquad \frac{\Gamma, z : T_1^z \vdash T_1^z <: T_2}{\Gamma \vdash \mu z.\ T_1^z <: T_2}\ \text{BindI}$$

$$\frac{\Gamma \vdash x : T^x}{\Gamma \vdash x : \mu z.\ T^z}\ \text{VarPack} \qquad \frac{\Gamma \vdash x : \mu z.\ T^z}{\Gamma \vdash x : T^x}\ \text{VarUnpack}$$

We adapt the notation employed by [41] for recursive types to our notation. In the rules, T^z denotes that variable z is free in type T. Rule BindX is in essence a one-step finite unfolding of the recursive type, leading to an inductive definition of subtyping. The second rule for recursive types (rule BindI) is a special case where a recursive type $\mu z.\ T_1^z$ is a subtype of another type T_2 if T_2 does not contain the recursive variable z. A difference between recursive types $\mu z.\ T^z$ in DOT and the ones in this paper is that in DOT z is a *term* variable instead of a type variable. In

DOT, recursive types can be used in combination with *path-dependent types* [3], to denote types such as $z.L$, where z represents a (possibly recursive) term with a type member L. Because of this design, the typing rules that introduce and eliminate recursive types [41], are defined on variables. Unlike our formulation of subtyping, which is algorithmic, DOT's formulation of subtyping is usually presented in a declarative form. Undecidability is an important problem with DOT's formulation of subtyping [27], and the existing decidable fragments of DOT lack transitivity [27,31].

Semantic Subtyping with Intersection Types and Equi-Recursive Types. Semantic subtyping [24,16], provides a set-theoretic point of view for type systems. In that approach, (equi-)recursive types and intersection types are interpreted as subsets of the model, the subtyping relation is decidable, and some important properties, such as transitivity, are derived naturally. Although semantic subtyping approaches have many advantages, they can be technically more involved, while the metatheory of syntactic formulations is simpler and generally easier to extend. Damm [20] explored a type system with equi-recursive types and intersection types. His subtyping relation is quite expressive, as it supports equi-recursive types and distributivity rules for subtyping. However, he does not follow the conventional syntactic formulations of subtyping, such as the one in this paper or those employed in DOT. Instead, types are encoded as regular tree expressions/set constraints. In contrast, our formulation is more conventional and supports iso-recursive types instead. Unlike our work Damm does not consider extensible records with concatenation.

Other Languages with Recursive Types and Intersection Types. Some languages employ recursive subtyping as well as intersection types, like Typescript [32] and Whiley [36]. Typescript has a rich type system but no formal document provided. [8] has formalized a subset of Typescript and proved that is safe, but unfortunately, no intersection types are supported. Whiley is an experimental language that supports recursive types, intersection types and subtyping. However, no work formalizes all the features together, lacking either recursive types [35] or intersection types [30].

6 Conclusion

Recursive types, extensible record types and intersection types are important features in many OOP languages, since object types can be modelled with recursive records and multiple inheritance can be modelled via intersection types [17] and record concatenation. Our λ_i^μ calculus illustrates that the 3 features can be put together in a single calculus and therefore can be used to provide simple encodings for objects. There are a few interesting directions for future work. One is to add polymorphism and bounded quantification to λ_i^μ, which is a significant feature for real world languages. Another one is to investigate distributive subtyping for iso-recursive subtyping. With distributivity, we can model a form of family polymorphism [23].

References

1. Abadi, M., Cardelli, L.: A theory of objects. Springer Science & Business Media (1996)
2. Alpuim, J., Oliveira, B.C.d.S., Shi, Z.: Disjoint polymorphism. In: European Symposium on Programming. pp. 1–28. Springer (2017)
3. Amin, N., Rompf, T., Odersky, M.: Foundations of path-dependent types. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications. OOPSLA '14, Association for Computing Machinery (2014)
4. Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A filter lambda model and the completeness of type assignment. The journal of symbolic logic **48**(4), 931–940 (1983)
5. Bi, X., Oliveira, B.C.d.S.: Typed first-class traits. In: 32nd European Conference on Object-Oriented Programming (ECOOP 2018) (2018)
6. Bi, X., Oliveira, B.C.d.S., Schrijvers, T.: The essence of nested composition. In: 32nd European Conference on Object-Oriented Programming (ECOOP 2018) (2018)
7. Bi, X., Xie, N., Oliveira, B.C.d.S., Schrijvers, T.: Distributive disjoint polymorphism for compositional programming. In: European Symposium on Programming. pp. 381–409. Springer, Cham (2019)
8. Bierman, G., Abadi, M., Torgersen, M.: Understanding typescript. In: European Conference on Object-Oriented Programming. pp. 257–281. Springer (2014)
9. Bruce, K., Cardeli, L., Castagna, G., Group, T.H.O., Leavens, G.T., Pierce, B.: On binary methods. Theory and Practice of Object Systems **1**(3) (1996)
10. Bruce, K.B., Cardelli, L., Pierce, B.C.: Comparing object encodings **155**(1/2), 108–133 (Nov 1999), `http://www.cis.upenn.edu/~bcpierce/papers/compobj.ps`
11. Cardelli, L.: A semantics of multiple inheritance. In: Semantics of Data Types (1984)
12. Cardelli, L.: Amber. In: LITP Spring School on Theoretical Computer Science. pp. 21–47. Springer (1985)
13. Cardelli, L.: Extensible records in a pure calculus of subtyping. Digital. Systems Research Center (1992)
14. Cardelli, L.: A language with distributed scope. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 286–297. POPL '95 (1995)
15. Cardelli, L., Mitchell, J.C.: Operations on records. Mathematical Structures in Computer Science **1**(1), 3–48 (1991)
16. Castagna, G., Frisch, A.: A gentle introduction to semantic subtyping. In: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming. pp. 198–199 (2005)
17. Compagnoni, A.B., Pierce, B.C.: Higher-order intersection types and multiple inheritance. Mathematical Structures in Computer Science (MSCS) **6**(5), 469–501 (1996)
18. Cook, W.R., Hill, W., Canning, P.S.: Inheritance is not subtyping. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 125–135. POPL '90 (1989)
19. Cook, W.R., Palsberg, J.: A denotational semantics of inheritance and its correctness. In: Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) (1989)

20. Damm, F.M.: Subtyping with union types, intersection types and recursive types. In: International Symposium on Theoretical Aspects of Computer Software. pp. 687–706. Springer (1994)
21. Davies, R., Pfenning, F.: Intersection types and computational effects. In: International Conference on Functional Programming (ICFP) (2000)
22. Dunfield, J.: Elaborating intersection and union types. Journal of Functional Programming **24**(2-3), 133–165 (2014)
23. Ernst, E.: Family polymorphism. In: European Conference on Object-Oriented Programming (ECOOP) (2001)
24. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 137–146. IEEE (2002)
25. Giarrusso, P.G., Stefanesco, L., Timany, A., Birkedal, L., Krebbers, R.: Scala step-by-step: Soundness for dot with step-indexed logical relations in iris. Proc. ACM Program. Lang. **4**(ICFP) (Aug 2020)
26. Glew, N.: An efficient class and object encoding. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 311–324. OOPSLA '00, Association for Computing Machinery (2000)
27. Hu, J.Z., Lhoták, O.: Undecidability of d¡: and its decidable fragments. Proceedings of the ACM on Programming Languages **4**(POPL), 1–30 (2019)
28. Huang, X., Oliveira, B.C.d.S.: A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In: 34th European Conference on Object-Oriented Programming (ECOOP 2020). vol. 166 (2020). `https://doi.org/10.4230/LIPIcs.ECOOP.2020.26`
29. Huang, X., Zhao, J., Oliveira, B.C.d.S.: Taming the merge operator: a type-directed operational semantics approach. Journal of Functional Programming (2021)
30. Jones, T., Pearce, D.J.: A mechanical soundness proof for subtyping over recursive types. In: Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs. pp. 1–6 (2016)
31. Mackay, J., Potanin, A., Aldrich, J., Groves, L.: Decidable subtyping for path dependent types. Proceedings of the ACM on Programming Languages **4**(POPL), 1–27 (2019)
32. Microsoft: Typescript (2021), `https://www.typescriptlang.org/`
33. Oliveira, B.C.d.S., Shi, Z., Alpuim, J.: Disjoint intersection types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 364–377 (2016)
34. Palsberg, J., Zhao, T.: Type inference for record concatenation and subtyping. Inf. Comput. **189**(1) (Feb 2004)
35. Pearce, D.J.: Rewriting for sound and complete union, intersection and negation types. ACM SIGPLAN Notices **52**(12), 117–130 (2017)
36. Pearce, D.J., Groves, L.: Whiley: a platform for research in software verification. In: International Conference on Software Language Engineering. pp. 238–248. Springer (2013)
37. Pierce, B.C.: Bounded quantification is undecidable. Information and Computation **112**(1), 131–165 (1994)
38. Pottier, F.: A 3-part type inference engine. In: European Symposium on Programming. pp. 320–335. Springer (2000)
39. Rémy, D.: A case study of typechecking with constrained types: Typing record concatenation (1995), presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK

40. Reynolds, J.C.: Preliminary design of the programming language forsythe (1988)
41. Rompf, T., Amin, N.: Type soundness for dependent object types (dot). In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 624–641 (2016)
42. The Coq Development Team: Coq (2019), https://coq.inria.fr
43. Wadler, P.: The expression problem (1998), discussion on the Java Genericity mailing list
44. Wang, F., Rompf, T.: Towards strong normalization for dependent object types (dot). In: 31st European Conference on Object-Oriented Programming (ECOOP 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
45. Zhang, W., Sun, Y., Oliveira, B.C.d.S.: Compositional programming. ACM Transactions on Programming Languages and Systems (TOPLAS) pp. 1–60 (2021)
46. Zhou, Y., Zhao, J., Oliveira, B.C.d.S.: Revisiting iso-recursive subtyping. Accepted at the ACM Transactions on Programming Languages and Systems (TOPLAS) (2022), https://i.cs.hku.hk/~ydzhou/static/TOPLAS21draft.pdf