JFP **31**, e28, 67 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press.
doi:10.1017/S0956796821000186

Taming the Merge Operator

XUEJING HUANG
Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong
(*e-mail:* xjhuang@cs.hku.hk)

JINXU ZHAO
Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong
(*e-mail:* jxzhao@cs.hku.hk)

BRUNO C. D. S. OLIVEIRA
Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong
(*e-mail:* bruno@cs.hku.hk)

Abstract

Calculi with *disjoint intersection types* support a *symmetric merge operator* with subtyping. The merge operator generalizes record concatenation to any type, enabling expressive forms of object composition, and simple solutions to hard modularity problems. Unfortunately, recent calculi with disjoint intersection types and the merge operator lack a (direct) operational semantics with expected properties such as *determinism* and *subject reduction*, and only account for *terminating programs*. This paper proposes a *type-directed operational semantics* (TDOS) for calculi with intersection types and a merge operator. We study two variants of calculi in the literature. The first calculus, called λ_i, is a variant of a calculus presented by Oliveira et al. (2016) and closely related to another calculus by Dunfield (2014). Although Dunfield proposes a direct small-step semantics for her calculus, her semantics lacks both determinism and subject reduction. Using our TDOS, we obtain a direct semantics for λ_i that has both properties. The second calculus, called λ_i^+, employs the well-known subtyping relation of Barendregt, Coppo and Dezani-Ciancaglini (BCD). Therefore, λ_i^+ extends the more basic subtyping relation of λ_i, and also adds support for record types and *nested composition* (which enables recursive composition of merged components). To fully obtain determinism, both λ_i and λ_i^+ employ a disjointness restriction proposed in the original λ_i calculus. As an added benefit the TDOS approach deals with recursion in a straightforward way, unlike previous calculi with disjoint intersection types where recursion is problematic. We relate the static and dynamic semantics of λ_i to the original version of the calculus and the calculus by Dunfield. Furthermore, for λ_i^+, we show a novel formulation of BCD subtyping, which is algorithmic, has a very simple proof of transitivity and allows for the modular addition of distributivity rules (i.e. without affecting other rules of subtyping). All results have been fully formalized in the Coq theorem prover.

1 Introduction

The *merge operator* for intersection types was first introduced in the Forsythe language over 30 years ago (Reynolds, 1988). It has since been studied, refined and used in some language designs by multiple researchers (Pierce, 1991; Castagna *et al.*, 1995; Dunfield, 2014; Oliveira *et al.*, 2016; Alpuim *et al.*, 2017; Bi *et al.*, 2018). At its essence, the merge operator allows the creation of values that can have *multiple types*, which are encoded as *intersection types* (Pottinger, 1980; Coppo *et al.*, 1981). For example, with the merge operator, the following program is valid:

$$\mathsf{let}\ x : \mathsf{Int}\ \&\ \mathsf{Bool} = 1\ ,,\ \mathsf{True}\ \mathsf{in}\ (x + 1,\ \mathsf{not}\ x)$$

Here the variable x has two types, expressed by the intersection type Int & Bool. The corresponding value for x is built using the merge operator (,,). Later uses of x, such as the expression $(x + 1,\ \mathsf{not}\ x)$ can use x both as an integer or as a boolean. For this particular example, the result of executing the expression is the pair (2, False).

The merge operator adds expressive power to calculi with intersection types. Much work on intersection types has focused on *refinement intersections* (Freeman & Pfenning, 1991; Davies & Pfenning, 2000; Dunfield & Pfenning, 2003), which only increase the expressive power of types. In systems with refinement intersections, types can simply be erased during the compilation. However, in those systems, the intersection type Int & Bool is invalid since Int and Bool are not refinements of each other. In other systems, including many OO languages with intersection types – such as Scala (Odersky *et al*., 2004), TypeScript (Microsoft, 2012), Flow (Facebook, 2014), or Ceylon (RedHat, 2011) – the type Int & Bool has no inhabitants and the simple program above is inexpressible. The merge operator adds expressiveness to terms and allows constructing values that inhabit the disjoint intersection type Int & Bool, enabling writing programs like the above.

There are various practical applications for the merge operator. As Dunfield (2014) argues, the merge operator provides "general mechanisms that subsume many different features". This is important because a new type system feature often involves extending the metatheory and implementation, which can be nontrivial. If instead we provide general mechanisms that can encode such features, adding new features will become a lot easier. Dunfield has illustrated this point by showing that *multi-field records*, *overloading* and forms of *dynamic typing* can all be easily encoded in the presence of the merge operator. Furthermore, when we restrict our attention to the concatenation of records, which the merge operator generalizes, the combination of record concatenation and subtyping paves the ground for encoding expressive forms of *multiple inheritance* (Wand, 1989; Rémy, 1995; Palsberg & Zhao, 2004; Zwanenburg, 1997).

More recently, the merge operator has been used in calculi with *disjoint intersection types* (Oliveira *et al*., 2016; Alpuim *et al*., 2017; Bi *et al*., 2018). The disjointness restriction means that the two values being merged cannot have conflicts. In such settings, the merge operator is *symmetric*, *associative*, and *commutative* (Bi *et al*., 2018). Such a variant of the merge operator has been used to encode several nontrivial object-oriented features, which enable highly dynamic forms of object composition not available in current mainstream languages such as Scala or Java. These include *first-class traits* (Bi & Oliveira, 2018), *dynamic mixins* (Alpuim *et al*., 2017), and forms of *family polymorphism* (Bi *et al*., 2018). These features enable widely used and expressive techniques for object composition used by JavaScript programmers (and programmers in other dynamically typed languages), but in a completely *statically type-safe* manner (Bi & Oliveira, 2018; Alpuim *et al*., 2017). For example, in the SEDEL language (Bi & Oliveira, 2018), which is based on disjoint intersection types, we can define and use first-class traits such as:

```
// addId takes a trait as an argument, and returns another trait
addId(base : Trait[Person], idNumber : Int) : Trait[Student] =
  trait inherits base ⇒ { // dynamically inheriting from an unknown person
      def id : Int = idNumber
  }
```

Similarly to classes in JavaScript, first-class traits can be passed as arguments, returned as results, and can be constructed dynamically (at runtime). In the program above, inheritance is encoded as a merge in the core language used by SEDEL.

Despite over 30 years of research, the semantics of the merge operator has proved to be quite elusive. In retrospect, this is perhaps not too surprising. It is well known that, in the closely related area of record calculi, the combination of record concatenation and subtyping is highly nontrivial (Cardelli & Mitchell, 1991). Since the merge operator for intersection types generalizes record concatenation and calculi with intersection types naturally give rise to subtyping, the semantics of the merge operator will clearly not be any simpler than the semantics of record concatenation with subtyping!

Because of its foundational importance, we would expect a simple and clear *direct* semantics for calculi with a merge operator. After all, this is what we get for other foundational calculi such as the *simply-typed lambda calculus*, *System F*, *System* F_ω, the *calculus of constructions*, *System* $F_{<:}$, *Featherweight Java* and others. All these calculi have a simple and elegant direct operational semantics, often presented in a small-step style (Wright & Felleisen, 1994). While for the merge operator there have been efforts in the past to define direct operational semantics, these efforts have placed severe limitations that disallow many of the previously discussed applications or they lacked important properties. Reynolds (1991) was the first to look at this problem, but in his calculus the merge operator is severely limited. Castagna *et al*. (1995) studied another calculus, where only merges of functions are possible. Pierce (1991) was the first to briefly consider a calculus with an unrestricted merge operator (called *glue* in his work). He discussed an extension to $F_\wedge$ with a merge operator but he did not study the dynamic semantics with the extension. Finally, Dunfield (2014) goes further and presents a direct operational semantics for a calculus with an unrestricted merge operator. However, the problem is that subject reduction and determinism are lost.

Dunfield also presents an alternative way to give the semantics for a calculus with the merge operator *indirectly by elaboration* to another calculus. This elaboration semantics is type-safe and offers, for instance, a reasonable implementation strategy, and it is also employed in more recent work on the merge operator with disjoint intersection types. However, the elaboration semantics has two major drawbacks. First, reasoning about the elaboration semantics is more involved: to understand the semantics of programs with the merge operator we have to understand the translation and semantics of the target calculus. This complicates informal and formal reasoning. Secondly, in calculi defined by elaboration, we want to have *coherence* (Reynolds, 1991), which is a property that ensures that the meaning of a program is not ambiguous. Dunfield's elaboration semantics is not coherent. To fix this, calculi with disjoint intersection types have to impose some restrictions. However, even with such restrictions, coherence comes at a high price: the calculi and proof techniques employed to prove coherence are complex and can only deal with terminating programs. The latter is a severe limitation in practice!

This paper proposes a *type-directed operational semantics* (TDOS) for calculi with intersection types and a merge operator. We study two calculi, which are variants of existing calculi with disjoint intersection types in the literature. The first calculus, called λ_i, is a variant of a calculus introduced by Oliveira *et al*. (2016), and it is also closely related to a calculus by Dunfield (2014). The second calculus, called λ_i^+, employs the well-known subtyping relation of Barendregt, Coppo and Dezani-Ciancaglini (BCD). BCD subtyping introduces distributivity of intersections over arrows in the subtyping relation:

$$\frac{\textsc{OS-distArr}}{(A \to B) \,\&\, (A \to C) \leqslant A \to B \,\&\, C}$$

From the point of view of coercive subtyping, this rule denotes that a merge of two functions can be converted to one function, if their input type is the same. Therefore, λ_i^+ extends the more basic subtyping relation of λ_i, and also adds support for record types and *nested composition*, subsuming the original λ_i^+ calculus by Bi *et al.* (2018).

Both calculi address two key difficulties in the dynamic semantics of calculi with a merge operator. The first difficulty is the type-dependent nature of the merge operator. Using type annotations in the TDOS to guide reduction (and influence operational behavior) addresses this difficulty, and paves the way to prove *subject reduction*. The second difficulty is that a fully unrestricted merge operator is inherently ambiguous. For instance the merge $1\,,,\,2$ can evaluate to both 1 and 2. Therefore, some restriction is still necessary for a deterministic semantics. To fully obtain determinism, both calculi employ a disjointness restriction that is used in calculi using disjoint intersection types, and two important new notions: *typed reduction* and *consistency*. Typed reduction is a reduction relation that can further reduce values under a certain type. Consistency is an equivalence relation on values, that is key for the determinism result. Determinism in TDOS offers the same guarantee that coherence offers in an elaboration semantics (both properties ensure that the semantics is unambiguous), but it is much simpler to prove. Additionally, the TDOS approach deals with recursion in a straightforward way, unlike λ_i and subsequent calculi (Bi *et al.*, 2018, 2019) where recursion is very problematic for proving coherence.

To further relate λ_i to the calculi by Dunfield and the original λ_i by Oliveira et al., we show two results. First, we show that the type system of λ_i is complete with respect to the original calculus. Second, the semantics of λ_i is sound with respect to an extension of Dunfield's semantics. The extension is needed because λ_i uses a slightly more powerful subtyping relation, which enables λ_i to account for merges of functions in a natural way compared to the original λ_i. Furthermore, for λ_i^+, we show a novel formulation of BCD subtyping, which is algorithmic, has a very simple proof of transitivity and allows for the modular addition of distributivity rules (i.e. without affecting other rules of subtyping). We also deal with several additional complications in the operational semantics that arise from the nested composition. The two calculi and their metatheory have been fully formalized in the Coq theorem prover.

In summary, the contributions of this paper are as follows:

- **The λ_i and λ_i^+ calculi and their TDOS:** We present a TDOS for two calculi with intersection types and a merge operator. The semantics of both calculi is *deterministic* and it has *subject reduction*.
- **Support for nonterminating programs:** Our new proof methods can deal with recursion, unlike the proof methods used in previous calculi with disjoint intersection types, due to limitations of the current proof approaches for coherence.
- **Typed reduction and consistency:** We propose the novel notions of typed reduction and consistency, which are useful to prove determinism and subject reduction.

- **Relation with other calculus with intersection types:** We relate λ_i with the calculi proposed by Dunfield and Oliveira et al. In short, all programs that are accepted by the original λ_i calculus can type-check with our type system, and the semantics of λ_i is sound with respect to Dunfield's semantics.
- **Novel algorithmic formulation of BCD subtyping:** In our new formulation, the challenging distributivity rules are added in a modular way, and the transitivity proof is straightforward.
- **Coq formalization:** All the results presented in this paper have been formalized in the Coq theorem prover and they are available from `https://github.com/XSnow/TamingMerge`.

This paper is an extended version from a conference paper (Huang & Oliveira, 2020). The λ_i^+ calculus and the algorithmic formulation of BCD subtyping are new. Furthermore, λ_i differs from the calculus in the conference paper (where it is called $\lambda_i^{;}$) in that it allows the top value to act like a function and employs *bidirectional typechecking* (Pierce & Turner, 1998). This change enables typing formulations (for both λ_i and λ_i^+) to be algorithmic. Moreover, we added an extra section giving more background and motivation.

Roadmap. Section 2 presents the background, motivation, and applications for the calculi. Section 3 gives an overview of the challenges and the design of the TDOS and introduces the specification of disjointness and consistency. Sections 4 and 5 present the type system and operational semantics, respectively. The relation with the original λ_i's type system, and with Dunfield's semantics, is discussed at the end of each section. Section 6 introduces a novel algorithmic formulation of BCD subtyping using *splittable types*. Section 7 presents the λ_i^+ calculus and its TDOS, which extends λ_i with BCD subtyping and records. The design of rules is significantly affected by the new distributive subtyping rules. Finally, Section 9 discusses the related work and Section 10 concludes.

2 Motivation and applications of the merge operator

A key advantage of the merge operator is its generality and the ability to model various programming language features. However, there are challenging problems that arise from the merge operator. In particular, the combination of the merge operator and subtyping is problematic. In this section, we revisit those challenges, as well as two applications of the merge operator: *typed first-class traits* (Bi & Oliveira, 2018) and *nested composition* (Bi *et al.*, 2018). For the simple examples illustrating the merge operator in Section 2.1, we assume that some convenience features not in our calculi (but supported, for instance, in SEDEL), including a let construct and some inference of type annotations.

2.1 The merge operator, ambiguity, and subtyping

Ambiguity. As we have discussed in Section 1, a key problem with the merge operator is ambiguity. The problem stems from the implicit (type-directed) extraction of values from merges. For instance, for the expression:

$$(1\,,,\,2) + 3$$

the result is ambiguous (it could be 4 or 5) since we could extract either 1 or 2 from the merge to add to 3. One way to avoid ambiguity is to restrict the types of merged values to have *disjoint* types (Oliveira *et al.*, 2016). For instance, Int is disjoint to Bool, so the merge $1 \,,, \mathsf{True}$ is accepted. In contrast, $1 \,,, 2$ is rejected. Disjointness leads to a symmetric, associative and commutative merge operator (Bi *et al.*, 2018). Other alternatives include having a biased merge operator, which allows conflicting values. In such a case, when extracting a value of a certain type from a merge, the merge is searched in a particular order (for instance left-to-right) and the first value of the searched type is returned (Dunfield, 2014).

The complications of subtyping. Intersection types naturally induce a subtyping relationship between types. However, subtyping and the subsumption rule enable a program to "forget" about some static information about the types of values. Since the extraction of values from merges is type-directed, such loss of type information can affect the search for the value. Consider the following program:

$$\mathsf{let}\ x : \mathsf{Bool} = \mathsf{True} \,,, 1\ \mathsf{in}\ (2 \,,, x) + 3$$

Note that here we view $\mathsf{True} \,,, 1$ as a value. The merge has type $\mathsf{Bool} \,\&\, \mathsf{Int}$, but because of subtyping it also has type Bool, the type for x. In a naive operational semantics, for the program above, we would eventually reach a point where we would need to extract a value from the merge $2 \,,, \mathsf{True} \,,, 1$. This merge has two conflicting integers values.

In a language employing a disjointness restriction the merge $2 \,,, \mathsf{True} \,,, 1$ ought to be rejected, but such a merge only appears at runtime. In the program itself, all merges are disjoint: $\mathsf{True} \,,, 1$ is disjoint; and $2 \,,, x$ is also disjoint since x has type Bool. Thus, the program should type-check! One possibility would be to abort the program at runtime with a disjointness error. However, this would defeat the main purpose of the disjointness restriction, which is to provide a way to *statically* prevent ambiguity.

A language offering an asymmetric merge operator would have other issues. Assuming that the merge operator would be right-biased (giving preference to the values on the right side), then a programmer may expect that because x has type Bool, $(2 \,,, x) + 3$ should evaluate to 5. However such *static reasoning* is not synchronized with the runtime behavior, since x contains the integer 1 and therefore the result of the evaluation would be 4.

From another perspective, we could expect that a valid optimization of the program above is to replace the expression $(2 \,,, x) + 3$ by $2 + 3$, since the static type of x has no integer type. This optimization would be valid (for both symmetric and asymmetric merges) if the origin of runtime values can be determined statically by looking at the types. However, this is not the case if we simply employ a naive semantics: we statically know that the merge contains an integer 2, but at runtime 1 is extracted instead. The issue is somewhat similar to the (naive) dynamic scoping semantics for the lambda calculus, where the origin of the values for free variables cannot be determined statically when a function is created. Static (or lexical) scoping solves the problem by using a more sophisticated semantics. Thus, a possible solution for the problem of determining the origin of values in merges statically in the presence of subtyping is to have a more sophisticated semantics as well.

Record concatenation and subtyping. The problems with the merge operator and subtyping are closely related to the problem of typing record concatenation in the presence of subtyping. The latter is well-acknowledged to be a difficult problem in the design of record calculi (Cardelli & Mitchell, 1991). Foundational work on programming languages in the end of the 80s and early 90s looked at this problem because the combination record concatenation with subtyping was perceived as a way to extend lambda calculi with support for OOP. In essence, since objects in OOP can be viewed as records, it is natural to look for a language that supports records. Furthermore, record concatenation would provide support for encoding *multiple inheritance*, which entails composing several objects/records together. Finally, subtyping is perceived as a key feature of OOP and should be supported as well. Unfortunately, the problem was found to be quite challenging, for very similar reasons to those that make the interaction of the merge operator with subtyping difficult. This should not come as a surprise, since the merge operator can generalize record concatenation. To see the relationship between the two problems, consider the following variant of the previous program with records:

$$\mathsf{let}\ x : \{n : \mathsf{Bool}\} \ = \ \{m = 1\} \,,, \{n = \mathsf{True}\}\ \mathsf{in}\ (\{m = 2\} \,,, x).m \ + 3$$

In this variant, x is a record with the static type $\{n : \mathsf{Bool}\}$, but having an extra field m that is hidden by subtyping. The record x is then merged with the record $\{m = 2\}$. Statically this merge seems safe, since the static types of both records do not share record labels in common. However, when doing the field lookup for m at runtime there would be two fields m with different values (once again assuming a naive semantics). In essence, we would have the same problems as with the earlier variant of the program without records.

As we have been hinting, a way to solve this problem is to change the operational semantics to account for types at runtime. We will discuss in depth the technical challenges and aspects of such an approach from Section 3 onwards. But before doing this, we first show why this is a problem worth solving in the first place, by illustrating interesting applications that can be defined in languages that support a merge operator in the presence of subtyping.

2.2 Typed first-class traits

To illustrate the interesting applications that a merge operator enables we briefly introduce *typed first-class traits* (Bi & Oliveira, 2018) in the SEDEL language. This application is not new to this paper, but it is useful to revisit it to illustrate the kinds of applications that are enabled by the merge operator. Typed first-class traits are very much in line with the applications that OOP researchers had in mind while seeking for calculi integrating record concatenation and subtyping. In particular, the merge operator naturally enables a form of multiple inheritance, as well as a powerful form of dynamic inheritance (where inherited implementations can be parameterized).

Traits (Schärli *et al.*, 2003) in Object-Oriented Programming provide a model of multiple inheritance. Both traits and *mixins* (Bracha & Cook, 1990; Flatt *et al.*, 1998) encapsulate a collection of related methods to be added to a class. The main difference between traits and mixins has to do with how conflicts are dealt with. Mixins use the order of composition to determine which implementation to pick in the case of conflicts. Traits require programmers to explicitly resolve the conflicts instead and reject compositions with conflicts. In essence, this difference is closely related to the choice of a symmetric or asymmetric model for the merge operator. Symmetric merges with disjoint intersection types are closely related to traits because merges with conflicts are rejected, and the composition is associative and commutative (just like the composition for traits). Asymmetric merges are closer to mixins, giving preference to one of the implementations in the case of conflicts. We point the reader to Schärli's et al. paper for an extensive discussion of the qualities of the trait model and a comparison with the mixin model.

The SEDEL language (Bi & Oliveira, 2018) has a variant of traits. It essentially adopts the original trait model, but traits in SEDEL are statically typed and support dynamic inheritance (unlike Schärli et al. traits). The semantics of SEDEL's traits is defined via an elaboration to a calculus with disjoint intersection types, where the merge operator is key to model trait composition. Our examples next are adapted from Bi and Oliveira.

A simple example of a trait in SEDEL is:

```
type Editor = {on_key : String → String, do_cut : String, show_help :
    String};
type Version = {version : String};

trait editor [self : Editor & Version] ⇒ {
  on_key(key : String) = "Pressing " ++ key;
  do_cut = self.on_key "C-x" ++ " for cutting text";
  show_help = "Version: " ++ self.version ++ " Basic usage..."
};
```

A trait can be viewed as a function taking a self argument and producing a record. In this example, the record, which contains three fields, is encoded as a merge of three single field records. Because all the fields have distinct field names, the merge is disjoint and the definition is accepted. Methods in SEDEL can be dynamically dispatched, as usual in OOP languages. For instance, in the trait `editor`, the `do_cut` method calls the `on_key` method via the self reference and it is dynamically dispatched. Moreover, traits in SEDEL have a self-type annotation similar to Scala (Odersky *et al*., 2004). In this example, the type of the self reference is the intersection of two record types `Editor` and `Version`. Note that `show_help` is defined in terms of an *undefined* `version` method. Usually, in a statically typed language like Java, an abstract method is required, making `editor` an abstract class. Instead, SEDEL encodes abstract methods via self-types. The requirements stated by the type annotation of self must be satisfied when later composing `editor` with other traits, i.e. an implementation of the method `version` should be provided.

First-class traits and dynamic inheritance. The interesting features in SEDEL are that traits are *first-class* and inheritance can be *dynamic*. The next example shows such features:

```
type Spelling = {check : String};

spell (base : Trait[Editor & Version, Editor]) =
  trait [self : Editor & Version] inherits base ⇒ {
    override on_key(key : String) = "Process " ++ key ++ " on spell editor";
    check = super.on_key "C-c"  ++ " for spelling check"
  };
```

The spell function takes a trait as an argument and returns a trait as a result. Thus, since traits can be passed as arguments and returned as results they are first-class (just like lambda functions in functional programming). The new trait adds a check method and overrides the on_key method of the base trait. The argument base is a trait of type Trait[Editor & Version, Editor], where the two types denote trait requirements and functionality respectively. As we can see from its definition, trait editor matches that type. Note that unlike mainstream OOP languages like Java, the inherited trait (which would correspond to a superclass in Java) is *parameterized*, thus enabling dynamic inheritance. In SEDEL the choice of the inherited trait (i.e. the superclass) can happen at runtime, unlike in languages with static inheritance (such as Java or Scala).

Multiple inheritance. Besides first-class traits and dynamic inheritance, multiple inheritance is also supported. The following trait illustrates multiple inheritance in SEDEL:

```
trait version ⇒ {
        version = "0.2"
};

trait spell_editor [self : Editor & Version & Spelling ]
  inherits spell editor & version ⇒ {};

editor1 = new[Editor & Version & Spelling] spell_editor;
```

The trait spell_editor inherits from both spell editor and version. The latter defines an implementation for the field version. Finally, an object editor1 can be created from the trait spell_editor.

2.3 Nested composition

The NeColus calculus (Bi *et al*., 2018) and the SEDEL language support *nested composition*. With nested composition, it is not only possible to compose top-level traits but also to compose any elements inside the top-level trait recursively. Nested composition enables simple solutions to hard modularity problems like the Expression Problem (Wadler, 1998), and it is enabled by a powerful form of subtyping where intersections can distribute over other constructs.

The expression problem. Here we present the SEDEL-style solution of the Expression Problem, originally described by Bi *et al*. (2018). The expression problem is a classic challenge about the extensibility of a programming language. In the expression problem, a data type of expressions is defined, with several cases (literals and additions in the following code) associated with some operations (e.g. evaluation). There are two directions to extend the data type: adding a new case and adding a new operation. In a solution both extensions should be independently defined, and it should be possible to combine them to close the diamond. In a typical OOP language, class inheritance makes it easy to add a new case to the data type, while extending in the other direction in a modular and type-safe way remains hard. To illustrate the SEDEL solution, we start from a simple language with two cases and one operation.

```
type IEval = { eval : Double };
type Lang  = { lit : Double → IEval, add : IEval → IEval → IEval };

trait implLang ⇒ {
  lit (x : Double)           = { eval = x };
  add (x : IEval) (y : IEval) = { eval = x.eval + y.eval }
} : Lang;
```

In the above example, two fields `lit` and `add` in `Lang` model the constructors for expressions. Trait `implLang` defines a concrete evaluation operation over expressions by providing implementations for `lit` and `add`. As observed by Bi et al., traits such as `implLang`, can be viewed as a family of related implementations in the sense of *family polymorphism* (Ernst, 2001). In family polymorphism the central idea is that classes can be nested inside other classes, to form a family of related classes. In `implLang`, we can view the trait itself as a family, and the implementations of the constructors as the implementations of the nested "classes". The type `Lang` can be understood as the type of the family.

Adding a new operation `print` is straightforward via a new trait `implPrint`, which is defined in a similar way to `implLang`.

```
type IPrint = { print : String };
type LangPrint = { lit : Double → IPrint, add : IPrint → IPrint → IPrint };

trait implPrint ⇒ {
   lit (x : Double)             = { print = x.toString };
   add (x : IPrint) (y : IPrint) = {
     print = "(" ++ x.print ++ " + " ++ y.print ++ ")"
   }
} : LangPrint;
```

Similarly, a new case for negation can be added independently. The type of the new trait is the intersection of `Lang` and a record type for negation. Correspondingly, its implementation also reuses `implLang` via trait inheritance.

```
type LangNeg = Lang & { neg : IEval → IEval };

trait implNeg inherits implLang ⇒ {
   neg (x : IEval) = { eval = 0 - x.eval }
} : LangNeg;
```

It is necessary to extend `implPrint` for the newly defined case `neg` before composing them.

```
trait implExt inherits implNeg & implPrint ⇒ {
   neg (x : IPrint) = { print= "-" ++ x.print }
};
```

The trait combines the missing method with the extension of two dimensions: `implNeg` and `implPrint`. The following code shows how we can use the extended arithmetic language.

```
type ExtLang = { lit : Double → IEval&IPrint, add : IEval&IPrint →
     IEval&IPrint → IEval&IPrint, neg : IEval&IPrint → IEval&IPrint };

fac = new[ExtLang] implExt;
e = fac.add (fac.neg (fac.lit 2)) (fac.lit 3);
main = e.print ++ " = " ++ e.eval.toString -- "(-2.0 + 3.0) = 1.0";
```

BCD subtyping and nested composition. Notably the expression `e` has type `IEval & IPrint` allowing both the `print` and `eval` methods to be called. This is possible because nested composition is triggered by the annotation `ExtLang` for `new` when creating the object `fac`. While the expression `implExt` has type `Trait[LangNeg & LangPrint & { neg : IPrint → IPrint }]`, the annotation in `new` forces the resulting object to have type `ExtLang`. This is allowed because with BCD-style subtyping (Barendregt *et al.*, 1983) the following subtyping statement holds:

```
LangNeg & LangPrint & { neg : IPrint → IPrint } <: ExtLang
```

In short, in BCD-style subtyping, intersections distribute over other type constructors, like functions or records, thus allowing the previous subtyping statement to hold. For instance, for records the distributivity rule for subtyping is

$$\frac{\textsc{os-distRcd}}{\{l : A\} \,\&\, \{l : B\} \leqslant \{l : A \,\&\, B\}}$$

Nested composition gives an operational meaning to such an upcast at runtime by suitably adapting the values of the subtype to the right form to fit the supertype. For a simple example, consider the merge of two records, each containing a field `x` of (disjoint) types `Int` and `String`, respectively. Because of the distributivity of intersections over records, such a merge can be typed as a single record, with a field `x` of type `Int&String`:

```
{x = 3},, {x = "Hello"} : {x : Int & String}
```

At runtime, the value `{x = 3},, {x = "Hello"}` will be converted into `{x = 3 ,, "Hello"}`, in order for the shape of the value to match up with the shape of the type. Since traits in SEDEL are in essence records of functions, distributivity can trigger a series of such transformations that reshape the values to match up with the shape of types. Such transformations are what we call nested composition, which is essentially reflecting the changes triggered by distributivity (and other subtyping) rules at runtime. Thus, the components that are nested inside the traits being composed are themselves recursively composed, enabling the creation of objects like `e` containing implementations for both `print` and `eval`.

Encoding source language features. The SEDEL language, used to illustrate the examples in this section, is built on top of the F_i calculus (Alpuim *et al.*, 2017). The F_i calculus is itself a polymorphic extension of the original version of λ_i (Oliveira *et al.*, 2016). In essence, source language constructs, such as traits and extensible records, are elaborated into the more primitive constructs available in F_i. In particular, traits are encoded using merges of single field records. To model self-references and inheritance, the elaboration into F_i employs ideas from Cook's denotational semantics of inheritance (Cook & Palsberg, 1989). The details of the elaboration of SEDEL's source language constructs into F_i are outside of the scope of this paper, and are given in work by Bi & Oliveira (2018). A more recent version of SEDEL, employing the same encoding, but targeting the F_i^+ calculus instead and enabling distributive subtyping and nested composition, was used by Bi *et al.* (2019). We have used such a version to present the examples in Section 2.2.

Although SEDEL itself cannot be fully elaborated to λ_i and λ_i^+, because of the absence of polymorphism, a large subset of it can indeed be directly encoded into λ_i and λ_i^+. In particular, all the examples that we present in Section 2.2 fit within this (monomorphic) subset of SEDEL. In fact, although SEDEL implementations support polymorphic traits, the encoding of traits presented by Bi & Oliveira (2018) only supports monomorphic traits for simplicity. Therefore, such an encoding of traits can essentially be directly applied to encode the examples in Section 2.2 into λ_i^+. One small point is that the encoding assumes a `letrec` construct, which needs to be encoded with fix points in λ_i and λ_i^+.

In addition to encoding source-level constructs, the type system of SEDEL also performs simple forms of type-inference, allowing source programs to be written with fewer annotations than those needed in λ_i and λ_i^+. During elaboration, the type information is used to insert extra type annotations, which are used by the core language.

3 An overview of the type-directed operational semantics

While the merge operator has many applications, designing a direct operational semantics for a calculus with the merge operator is not straightforward. This section gives an overview of the type-directed operational semantics for λ_i and λ_i^+. We first introduce the untyped semantics of Dunfield (2014) to show the behavior of the merge operator. As Dunfield herself noted, such a semantics has two important problems: *nondeterminism*; and the *lack of subject reduction*. In this section we illustrate how the TDOS uses type annotations to guide reduction, solving the problems in Dunfield's semantics.

3.1 Background: Dunfield's nondeterministic semantics

Dunfield studied a calculus with unrestricted intersections and unions. The interesting aspect of her calculus is the merge operator: it takes two terms E_1 and E_2 of some types A and B, to create a new term that can behave both as a term of type A and as a term of type B. She proposed an operational semantics and an indirect elaboration semantics for the calculus. As Dunfield noted, the operational semantics is nondeterministic and does not preserve types. Nonetheless, such a semantics plays an important role in Dunfield's work by giving an over-approximation of the intended behavior of programs using the merge operator. It is used to justify the elaboration semantics, where programs are compiled into a language with products and sums, and without intersections and unions. Indeed, intersection types and the merge operator in Dunfield's calculus are similar to product types and pairs. For example, a program written with pairs that behaves identically to the program shown in Section 1 is:

$$\mathsf{let}\; x : (\mathsf{Bool}, \mathsf{Int}) = (\mathsf{True}, 1)\; \mathsf{in}\; (\mathsf{snd}\;\; x + 1, \mathsf{not}\,(\mathsf{fst}\;\; x))$$

However while for pairs both the introductions and eliminations are explicit, with the merge operator the eliminations (i.e. projections) are *implicit* and driven by the types of the terms. Dunfield exploits this similarity in her elaboration semantics. By extending typing judgments, the elaboration transforms merges into pairs, intersections into products and inserts the missing projections.

Syntax. The top of Figure 1 shows the syntax of Dunfield's calculus. Types include a top type Top, function types ($A \to B$) and intersection types (written as $A \,\&\, B$). Following the

(Syntax of Dunfield's Calculus)

Types	A, B	$::=$	$\mathsf{Top} \mid A \to B \mid A \,\&\, B$
Expressions	E	$::=$	$x \mid \top \mid \lambda x.\, E \mid E_1\, E_2 \mid \mathsf{fix}\, x.\, E \mid E_1 \mathbin{,,} E_2$
Values	V	$::=$	$x \mid \top \mid \lambda x.\, E \mid V_1 \mathbin{,,} V_2$

$\boxed{E \leadsto E'}$ *(Operational Semantics of Dunfield's Calculus)*

$$\textsc{DStep-appl}\ \frac{E_1 \leadsto E_1'}{E_1\, E_2 \leadsto E_1'\, E_2} \qquad \textsc{DStep-appr}\ \frac{E_2 \leadsto E_2'}{V_1\, E_2 \leadsto V_1\, E_2'} \qquad \textsc{DStep-beta}\ \frac{}{(\lambda x.\, E)\; V \leadsto E[x \mapsto V]}$$

$$\textsc{DStep-fix}\ \frac{}{\mathsf{fix}\, x.\, E \leadsto E[x \mapsto \mathsf{fix}\, x.\, E]} \qquad \textsc{DStep-mergel}\ \frac{E_1 \leadsto E_1'}{E_1 \mathbin{,,} E_2 \leadsto E_1' \mathbin{,,} E_2} \qquad \textsc{DStep-merger}\ \frac{E_2 \leadsto E_2'}{V_1 \mathbin{,,} E_2 \leadsto V_1 \mathbin{,,} E_2'}$$

$$\textsc{DStep-unmergel}\ \frac{}{E_1 \mathbin{,,} E_2 \leadsto E_1} \qquad \textsc{DStep-unmerger}\ \frac{}{E_1 \mathbin{,,} E_2 \leadsto E_2} \qquad \textsc{DStep-split}\ \frac{}{E \leadsto E \mathbin{,,} E}$$

Fig. 1. Syntax and nondeterministic small-step semantics of Dunfield's calculus.

convention introduced by previous works (Oliveira *et al.*, 2016), $\to$ has lower precedence than &, which means $A \to B \,\&\, C$ is equal to $A \to (B \,\&\, C)$. Most expressions are standard, except for the merge $E_1 \mathbin{,,} E_2$. The calculus also includes a canonical top value $\top$, and considers variables as values. Note that the original calculus by Dunfield uses a different notation for intersection types ($A \wedge B$), and supports union types ($A \vee B$). For a better comparison, we adjust the syntax and omit union types in Dunfield's system. Union types are not supported by λ_i, since it is based on the calculus by Oliveira *et al.* (2016) with disjoint intersection types, which focus on the merge operator and does not have unions types. Integrating union types with disjointness is still an open problem, which has not been addressed in previous work on disjoint intersection types. Since such a problem is orthogonal to the work in this paper, the calculi presented in this paper do not support union types either.

Operational semantics. The bottom part of Figure 1 presents the reduction rules. All rules not involving the merge operator are standard call-by-value reduction rules. The reduction of a merge construct in Dunfield's calculus is quite flexible: $V_1 \mathbin{,,} V_2$ is a value, but $E_1 \mathbin{,,} E_2$ (including values) can step, to its left subexpression (by rule DStep-unmergel) or the right one (by rule DStep-unmerger). Other values can step further as well, by rule DStep-split, which allows any expression to split into two.

Problem 1: no subject reduction. The operational semantics does not preserve types. Since the reduction is oblivious of types, a term can reduce to two terms with potentially different (and unrelated) types. For instance,

$$1 \mathbin{,,} \mathsf{True} \leadsto 1 \qquad\qquad 1 \mathbin{,,} \mathsf{True} \leadsto \mathsf{True}$$

Here the merge of an integer and a boolean is reduced to either the integer (rule DStep-unmergel) or the boolean (rule DStep-unmerger). In Dunfield's calculus the term $1 \mathbin{,,} \mathsf{True}$ can have multiple types, including Int or Bool or $\mathsf{Int} \,\&\, \mathsf{Bool}$. Not all types that can be assigned to a term lead to type-preserving reductions. For instance, if the term is given the type Int, then the second reduction above does not preserve the type. What is worse, a well-typed expression can reduce to an ill-typed expression by dropping the wrong part:

$$(1 ,, \lambda x.\, x + 1)\, 2 \leadsto 1\, 2$$

Problem 2: nondeterminism. Even in type-preserving reductions there can be another problem. Because of the pair of UNMERGE rules (rule DSTEP-UNMERGEL and rule DSTEP-UNMERGER), the choice between a merge always has two options. This means that a reduced term can lead to two other terms of the same type, but with different meanings. For example,

$$1 ,, 2 \leadsto 1 \qquad 1 ,, 2 \leadsto 2$$

There is even a third option to reduce a merge with the split rule (rule DSTEP-SPLIT):

$$1 ,, 2 \leadsto (1 ,, 2) ,, (1 ,, 2)$$

In other words, the semantics is nondeterministic. Nondeterminism is also the root of the problem with the examples discussed in Section 2.1.

Dunfield's elaboration semantics, although is type-safe, also suffers from this issue. The merge $1 ,, 2$ can elaborate to 1 or 2 when checked against Int by the typing rules. Her implementation prioritizes the left part, resulting in a biased merge operator.

3.2 A type-driven semantics for type preservation

An essential problem is that the semantics cannot ignore the types if the reduction is meant to be type-preserving. Dunfield (2014) notes that "*For type preservation to hold, the operational semantics would need access to the typing derivation*". To avoid runtime type-checking, we design a type-driven semantics and use type annotations to guide reduction. Therefore, our λ_i calculus is explicitly typed, unlike Dunfield's calculus. Nevertheless, it is easy to design source languages that infer some of the type annotations and insert them automatically to create valid λ_i terms as we will see in Section 4.4. We discuss the main challenges and key ideas of the design of λ_i next.

Annotations and type-driven reduction. Our operational semantics follows a standard call-by-value small-step reduction and it is closely related to Dunfield's semantics. However, type annotations are used to guide reduction. For example, in λ_i we can write explicitly annotated expressions for which the following reductions are valid:

$$(1 ,, \mathsf{True}) : \mathsf{Int} \hookrightarrow 1 \qquad (1 ,, \mathsf{True}) : \mathsf{Bool} \hookrightarrow \mathsf{True}$$

In contrast, the following reductions are not possible:

$$(1 ,, \mathsf{True}) : \mathsf{Bool} \not\hookrightarrow 1 \qquad (1 ,, \mathsf{True}) : \mathsf{Int} \not\hookrightarrow \mathsf{True}$$

Note also that $1 ,, \mathsf{True}$ without any type annotation is a value and does not reduce.

Typed reduction in action. The crucial component in the operational semantics, which enables the use of type information during reduction, is an auxiliary *typed reduction* relation $v \hookrightarrow_A v'$. This relation is used when we want some value to match a type. Typed reduction is where type information from annotations in λ_i "filters" reductions that are invalid due to a type mismatch. It takes a value and a type as inputs and produces a value of that type as output. Similarly, to Dunfield's operational semantics, this process may result in further reduction of values, unlike many other languages where values can never be further reduced. Typed reduction is used in two places during reduction:

$$\textsc{Step-annov}\ \frac{v \hookrightarrow_A v'}{v : A \hookrightarrow v'} \qquad \textsc{Step-beta}\ \frac{v \hookrightarrow_A v'}{(\lambda x.\, e : A \to B)\, v \hookrightarrow (e[x \mapsto v']) : B}$$

The first place is in rule Step-annov. When reduction meets a value v with a type annotation A, it uses typed reduction to further reduce v against the type A. To see typed reduction in action, consider a simple merge of primitive values $1 \,,, \mathsf{True} \,,, \text{'}c\text{'}$ with an annotation $\mathsf{Int} \,\&\, \mathsf{Char}$. Using rule Step-annov, typed reduction is invoked, resulting in:

$$1 \,,, \mathsf{True} \,,, \text{'}c\text{'} \hookrightarrow_{\mathsf{Int}\,\&\,\mathsf{Char}} 1 \,,, \text{'}c\text{'}$$

We can also type-reduce the same value under a similar type but where the two types in the intersection are interchanged:

$$1 \,,, \mathsf{True} \,,, \text{'}c\text{'} \hookrightarrow_{\mathsf{Char}\,\&\,\mathsf{Int}} \text{'}c\text{'} \,,, 1$$

The two valid reductions illustrate the ability of typed reduction to create a value that matches exactly with the shape of the type.

The second place where typed reduction is used is in rule Step-beta. In a function application, the argument can contain more components than what the function expects. That is because subtyping is allowed for function inputs. For intersection types, we know $A \,\&\, B <: A$. Thus, a function can take an input of type $A \,\&\, B$ while it expects an A. One concrete example is $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\,(1 \,,, \mathsf{True})$. The merge term $(1 \,,, \mathsf{True})$ provides an integer 1, while the other component True is useless. Such redundant components are sometimes even harmful. In the following example, we directly substitute the argument to demonstrate why we need to do typed reduction first.

$$(\lambda x.\, (x \,,, \mathsf{False}) : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool})\,(1 \,,, \mathsf{True}) \hookrightarrow (1 \,,, \mathsf{True} \,,, \mathsf{False}) : \mathsf{Int} \,\&\, \mathsf{Bool}$$

The function expects an integer but is applied to $(1 \,,, \mathsf{True})$. Direct substitution (as done usually by beta-reduction) would lead to a merge containing two different booleans: $1 \,,, \mathsf{True} \,,, \mathsf{False}$. Such an ambiguous term is not well typed, as we shall see in Section 3.4. Thus, such reduction would not be type-preserving.

To avoid the issue with direct substitution, we choose to drop the boolean before beta reduction by type reducing the argument. The input type annotation of the applied function is used to guide the typed reduction and therefore cannot be omitted in lambda expressions. With only one boolean, the final result is well typed.

$$(\lambda x.\, (x \,,, \mathsf{False}) : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool})\,(1 \,,, \mathsf{True}) \hookrightarrow (1 \,,, \mathsf{False}) : \mathsf{Int} \,\&\, \mathsf{Bool}$$

Note that, even an identity function could change its input value.

$$(\lambda x.\, x : \mathsf{Int} \to \mathsf{Int})\,(1 \,,, \text{'}c\text{'}) \hookrightarrow 1 : \mathsf{Int}$$

That may look strange from the view of the subtyping models used in conventional OOP languages, where upcasting has no runtime impact. However, in coercive subtyping (Luo, 1999), subtyping triggers coercions, and such coercions can have an operational effect and change the underlying value. Previous work on intersection types with the merge operator (Dunfield, 2014; Oliveira *et al.*, 2016; Alpuim *et al.*, 2017; Bi *et al.*, 2018, 2019) employ an elaboration semantics with coercive subtyping: the elaboration introduces coercions that transform values. For example, a merge is translated into a pair, and the coercion from $\mathsf{Int}\ \&\ \mathsf{Char}$ to Int is a projection on the pair which takes the first component. In our semantics, typed reduction plays a similar role to coercions, and our model of subtyping is quite close to coercive subtyping. Annotations trigger typed reduction, which plays a role similar to coercions in an elaboration approach.

The examples in this (sub)section have shown some nontrivial aspects of typed reduction, which must decompose values, and possibly drop (or rather hide via subtyping) some of the components and permute other components. The details of the typed reduction relation will be discussed in Section 5. As we shall see next functions introduce further complications.

3.3 The challenges of functions

Some of the hardest challenges in designing the semantics of λ_i involve functions. We discuss them next.

Return types matter. Unlike primitive values, we cannot tell the type of a function by its form. Although the input type annotation of lambdas helps beta reduction, it is not enough to distinguish among multiple functions in a merge (e.g. $(\lambda x.\, x + 1)\ ,,\ (\lambda x.\, \mathsf{True})$) without runtime type-checking. To be able to select the right function from a merge, in λ_i, all functions are annotated with both the input and output types. With such annotations, we can deal with programs like:

$$((\lambda f.\, f\ 1 : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}))\ ((\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\ ,,\ (\lambda x.\, \mathsf{True} : \mathsf{Int} \to \mathsf{Bool}))$$

In this program, we have a lambda that takes a function f as an argument and applies it to 1. The lambda is applied to the merge of two functions of types $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Int} \to \mathsf{Bool}$. We select the wanted function by comparing their type annotations to the target type in typed reduction. Otherwise, runtime type-checking would be necessary to recover the full type of functions.

Annotation refinement. Typed reduction reduces any value v to another value v' that has a supertype of the type of v. Subtyping of intersection types leads to selecting and dropping components from merges. On the other hand, the subtyping of arrow types requests for type refinements on lambda expressions. Consider a single function $\lambda x.\, x\ ,,\ \mathsf{False} : \mathsf{Int} \to \mathsf{Int}\ \&\ \mathsf{Bool}$ to be reduced under type $\mathsf{Int}\ \&\ \mathsf{Bool} \to \mathsf{Int}$. To let the function return an integer when applied to a merge of type $\mathsf{Int}\ \&\ \mathsf{Bool}$, we must change either the lambda body or the embedded annotation. Since reducing under a lambda body is not allowed in call-by-value, λ_i adopts the latter option, and treats the input and output annotations differently. The input annotation should not be changed as it represents the expectation of the function and helps to adjust the input value before substitution. The next example demonstrates that refining the input type annotation could result in ambiguity (having both True and False in one merge), which we prevent using rule STEP-BETA.

$$\begin{array}{ll}
 & (\lambda x.\, x \,,,\, \mathsf{False} : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool} : \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int})\,(1 \,,,\, \mathsf{True}) \\
\hookrightarrow & \{ \text{ wrong step here: not keeping the input annotation in typed reduction } \} \\
 & (\lambda x.\, x \,,,\, \mathsf{False} : \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int})\,(1 \,,,\, \mathsf{True}) \\
\hookrightarrow & \{ \text{ by rule STEP-BETA } \} \\
 & (1 \,,,\, \mathsf{True} \,,,\, \mathsf{False}) : \mathsf{Int} \qquad\qquad \text{(Does not type check!)}
\end{array}$$

The output annotation, in contrast, must be replaced by Int, representing a future reduction to be done after substitution. The output of the application then can be thought of as an integer and can be safely merged with another boolean. The next example illustrates how λ_i correctly deals with annotation refinements:

$$\begin{array}{ll}
 & ((\lambda x.\, x \,,,\, \mathsf{False} : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool} : \mathsf{Int} \,\&\, \mathsf{Bool} \to \mathsf{Int})\,(1 \,,,\, \mathsf{True})) \,,,\, \mathsf{True} \\
\hookrightarrow & \{ \text{ keep the input annotation and change the output one } \} \\
 & ((\lambda x.\, x \,,,\, \mathsf{False} : \mathsf{Int} \to \mathsf{Int})\,(1 \,,,\, \mathsf{True})) \,,,\, \mathsf{True} \\
\hookrightarrow & \{ \text{ by rule STEP-BETA } \} \\
 & (1 \,,,\, \mathsf{False}) : \mathsf{Int} \,,,\, \mathsf{True} \\
\hookrightarrow & \{ \text{ by rule STEP-ANNOV } \} \\
 & 1 \,,,\, \mathsf{True}
\end{array}$$

This example is similar to the previous one but, additionaly, we merge the expression with True to demonstrate that the ouput type after beta-reduction, will filter the resulting merge.

Some calculi avoid the problem of function annotation refinement by treating annotated lambdas as values. For example, the target language of NeColus does not reduce a value wrapped by a coercion in a function form. In the blame calculus (Wadler & Findler, 2009), a value with a cast from an arrow type to another arrow type is still a value.

3.4 Disjoint intersection types and consistency for determinism

Even if the semantics is type directed and it rules out reductions that do not preserve types, it can still be nondeterministic. To solve this problem, we employ the disjointness restriction that is used in calculi with disjoint intersection types (Oliveira *et al.*, 2016) and the novel notion of *consistency*. Both disjointness and consistency play a fundamental role in the proof of determinism.

Disjointness. Two types are disjoint (written as $A * B$), if any common supertypes that they have are *top-like types* (i.e. supertypes of any type; written as $\rceil C \lceil$).

Definition 3.1 (Disjoint types). $A * B \equiv \forall C \textit{ if } A <: C \textit{ and } B <: C \textit{ then } \rceil C \lceil$

If two types are disjoint (e.g. (Int & Char) $*$ Bool), their corresponding values do not overlap (e.g. 1 ,, 'c' and True). The only exceptions are top-like types, as they are disjoint with any type (Alpuim *et al.*, 2017). Since every value of a top-like type has the same effect, typed reduction unifies them to a fixed result. Thus the disjointness check in the following typing rule guarantees that e_1 and e_2 can be merged safely, without any ambiguities. For example, this typing rule does not accept $1 ,, 2$ or $\mathsf{True} ,, 1 ,, \mathsf{False}$, as two subterms of the merge have overlapped types (in this case, the same type Int and Bool, respectively).

$$\textsc{Typ-merge}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A * B}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A \,\&\, B}$$

Note that in this rule, $\Rightarrow$ denotes the *synthesis mode* in *bidirectional typing*. In typing judgements with such a mode, types are synthesized from the term, rather than checked.

Consistency. Recall the rule DStep-split in Dunfield's semantics: $E \rightsquigarrow E \;,,\; E$. It duplicates terms in a merge. Similar things can happen in our typed reduction if the type has overlapping parts, which is allowed, for example, in an expression $1 : \mathsf{Int} \,\&\, \mathsf{Int}$. Note that in this expression the term 1 can be given type annotation $\mathsf{Int} \,\&\, \mathsf{Int}$ since $\mathsf{Int} <: \mathsf{Int} \,\&\, \mathsf{Int}$. During reduction, typed reduction is eventually used to create a value that matches the shape of type $\mathsf{Int} \,\&\, \mathsf{Int}$ by duplicating the integer:

$$1 \hookrightarrow_{\mathsf{Int}\,\&\,\mathsf{Int}} 1 ,, 1$$

Note that the disjointness restriction does not allow sub-expressions in a merge to have the same type: $1 ,, 1$ cannot type-check with rule Typ-merge. To retain *type preservation*, there is a special typing rule for merges of values, where a novel consistency check is used (written as $v_1 \approx_{spec} v_2$):

$$\textsc{Typ-mergev}\quad \frac{\cdot \vdash v_1 \Rightarrow A \qquad \cdot \vdash v_2 \Rightarrow B \qquad v_1 \approx_{spec} v_2}{\Gamma \vdash v_1 ,, v_2 \Rightarrow A \,\&\, B}$$

This rule is not designed to accept more programs written by users. Instead, it takes care of expressions like $1 ,, 1$ that may appear at runtime. Mainly, consistency allows values to have overlapped parts as far as they are syntactically equal. For example, $1 ,, \mathsf{True}$ and $1 ,, `c'$ are consistent, since the overlapped part $\mathsf{Int}$ in both of merges has the same value. $\mathsf{True}$ and $`c'$ are consistent because they are not overlapped at all. But $1 ,, \mathsf{True}$ and 2 are *not consistent*, as they have different values for the same type Int. When two values have disjoint types, they must be consistent. For merges of such values, both rule Typ-mergev and rule Typ-merge can be applied, and the types always coincide. In λ_i, consistency is defined in terms of typed reduction:

Definition 3.2 (Consistency)**.** *Two values v_1 and v_2 are said to be consistent (written $v_1 \approx_{spec} v_2$) if, for any type A, the result of typed reduction for the two values is the same.*

$$v_1 \approx_{spec} v_2 \equiv \forall A \text{ if } v_1 \hookrightarrow_A v_1' \text{ and } v_2 \hookrightarrow_A v_2' \text{ then } v_1' = v_2'$$

Although the specification of consistency is decidable and an equivalent algorithmic definition exists (later defined in Figure 14), an algorithmic definition is not required. In practice, in a programming language implementation, the rule Typ-mergev may be omitted, since, as stated, its main purpose is to ensure that runtime values are type-preserving. On the other hand, after preservation is proved in metatheory, runtime values are guaranteed to be well typed and therefore there is no need to employ runtime type-checking.

Note that the original λ_i (Oliveira *et al.*, 2016) is stricter than our variant of λ_i and forbids any intersection types which are not disjoint. That is to say, the term $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is not well-typed because the intersection $\mathsf{Int} \,\&\, \mathsf{Int}$ is not disjoint. In the original λ_i calculus disjointness checking is done by defining type well-formedness and forbidding all intersections of two nondisjoint types. However, this approach is more conservative and less expressive.

The idea of allowing unrestricted intersections, while only having the disjointness restriction for merges, was first employed in the NeColus calculus (Bi *et al.*, 2018). λ_i follows such an idea and $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is well-typed in λ_i. Allowing unrestricted intersections adds extra expressive power. For instance, in calculi with polymorphism, unrestricted intersections can be used to encode *bounded quantification* (Cardelli & Wegner, 1985), whereas with disjoint intersections only such an encoding does not work (Bi *et al.*, 2019; Xie *et al.*, 2020). Various authors, including Pierce and Castagna, have (informally) observed that some form of bounded quantification can be encoded via (unrestricted) intersection types (Pierce, 1991; Castagna & Xu, 2011). Xie *et al.* (2020) formalize this encoding precisely. For further details on this encoding, as well as to why unrestricted intersections are needed, we refer to the work of Xie et al.

Revisiting the examples with merges and subtyping. Recall the first example presented in Section 2, rewritten here to use a lambda instead of a let expression:

$$(\lambda x.\,(2\,,,\,x)\;+3 : \mathsf{Bool} \to \mathsf{Int})\,(\mathsf{True}\,,,\,1)$$

As argued in Section 2, in a naive untyped semantics, examples like the above are problematic since they can lead to nondisjoint merges appearing at runtime. So, how does the TDOS approach deal with such an example? Here are the full reduction steps:

$$\begin{array}{lll}
 & (\lambda x.\,(2\,,,\,x)\;+3 : \mathsf{Bool} \to \mathsf{Int})\,(\mathsf{True}\,,,\,1) & \\
\hookrightarrow & ((2\,,,\,\mathsf{True})\;+3) : \mathsf{Int} & \text{by rule STEP-BETA} \\
\hookrightarrow & 5 : \mathsf{Int} & \text{reduction for } + \\
\hookrightarrow & 5 & \text{by rule STEP-ANNOV}
\end{array}$$

First, the input value is filtered via the typed reduction against the input type Bool. Importantly, *only* the selected part True is substituted in the body of the lambda during beta-reduction. Then, the expression $(2\,,,\,\mathsf{True})\;+3$ evaluates to 5. In the process, $+$ acts like a lambda with annotation $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$. Finally, the return-type Int filters the result, which does not change it in this case.

The second example with records is slightly more involved, but can be dealt with similarly. In the example with records, besides the use of the let expressions, which are not present in λ_i or λ_i^+, we assume some type-inference that avoids some explicit type annotations. Therefore, to encode such an example in λ_i^+, we must first introduce an explicit type annotation. The SEDEL language, in its elaboration process to the F_i core language (which is similar to λ_i or λ_i^+, except for the addition of polymorphism), inserts such annotations automatically (see also Section 8.3).

We show the full reduction steps, starting from the original example encoded as a λ_i^+ expression using a lambda instead of a let expression, and with an extra annotation $(\{m : \mathsf{Int}\})$ in Figure 2. In this example, a function that expects a record of Bool takes a

$$
\begin{array}{ll}
& (\lambda x.\ ((\{m = 2\} \,,,\, x) : \{m : \mathsf{Int}\}).m\ + 3 : \{n : \mathsf{Bool}\} \to \mathsf{Int})\ (\{m = 1\} \,,,\, \{n = \mathsf{True}\}) \\
\hookrightarrow & \{ \text{ by rule } \textsc{Step-beta} \text{ and typed reduction} \} \\
& (((\{m = 2\} \,,,\, \{n = \mathsf{True}\}) : \{m : \mathsf{Int}\}).m\ + 3) : \mathsf{Int} \\
\hookrightarrow & \{ \text{ by rule } \textsc{Step-anno} \ \} \\
& (\{m = 2\}.m\ + 3) : \mathsf{Int} \\
\hookrightarrow & \{ \text{ reduction for record projection } \} \\
& (2\ + 3) : \mathsf{Int} \\
\hookrightarrow & \{ \text{ reduction for } + \ \} \\
& 5 : \mathsf{Int} \\
\hookrightarrow & \{ \text{ by rule } \textsc{Step-annov} \ \} \\
& 5
\end{array}
$$

Fig. 2. Reduction for the record example.

merge of two records. Only the one with the expected label and field is selected via typed reduction. Later, the annotation $\{m : \mathsf{Int}\}$ helps to drop the record with unmatched labels before projection. Afterward, the reduction is straightforward.

3.5 The challenges of distributivity

The λ_i calculus captures the basic functionality of the merge operator with a simple subtyping relation with intersection types. However, such a simple subtyping relation lacks distributivity rules that enable, for instance, subtyping statements such as

$$(A \to B) \,\&\, (C \to D) <: A \,\&\, C \to B \,\&\, D$$

where the intersections distribute over the function types. Distributivity in a calculus with a merge operator is interesting because it enables nested composition, which essentially reflects the distributivity seen at the type level into the term level. Therefore, a merge of two functions can be treated as a single function where the inputs and outputs of the two original functions have intersection types.

The λ_i^+ calculus extends λ_i with distributivity rules for subtyping and nested composition. The subtyping relation for λ_i^+ is based on the well-known subtyping relation of Barendregt, Coppo and Dezani-Ciancaglini (1983). Adding BCD style subtyping into the type system of λ_i enables interesting applications, but it also brings more challenges.

Splittable arrow types. Without distributivity, if an arrow type is a supertype of an intersection of multiple types, then it must be a supertype of one of those types. Conversely, when doing typed reduction of a merge under an arrow type, we will obtain a single function (one of the components of the merge) as a result. However, in λ_i^+, we are now faced with the following kind of typed reduction due to the change in subtyping:

$$
\begin{array}{rl}
& (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) \,,,\, (\lambda x.\, \mathsf{True} : \mathsf{Int} \to \mathsf{Bool}) \,,,\, (\lambda x.\, \text{`}c\text{'} : \mathsf{Int} \to \mathsf{Char}) \\
\hookrightarrow_{\mathsf{Int} \to \mathsf{Int}\,\&\,\mathsf{Char}} & (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) \,,,\, (\lambda x.\, \text{`}c\text{'} : \mathsf{Int} \to \mathsf{Char})
\end{array}
$$

Even though we are doing typed reduction under an arrow type, we do not obtain a function as a result. Instead what we have is a merge of two functions. This is because, with the distributivity of arrow types, multiple components present in a merge can contribute to the final result. For instance, in the reduction above both the first and the last lambdas must be present to ensure that the resulting value "behaves" as a function of type $\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Char}$.

Parallel application. One consequence of allowing merges to have arrow types is that a beta reduction for applications is not enough, since merges of functions can also be applied to values. We use a relation called the parallel application to deal with applications of merges to another value. Parallel application distributes the input to every lambda in a merge, and beta reduces them in parallel. From the point of view of the small-step semantics, the parallel application process, like typed reduction, is finished in a single step, like the following example.

$$\begin{aligned} &(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int} \,,,\, (\lambda x.\, \mathsf{True} : \mathsf{Int} \to \mathsf{Bool} \,,,\, \lambda x.\, `c\text{'} : \mathsf{Int} \to \mathsf{Char}))\; 2 \\ \hookrightarrow\; &(2 + 1) : \mathsf{Int} \,,,\, (\mathsf{True} : \mathsf{Bool} \,,,\, `c\text{'} : \mathsf{Char}) \end{aligned}$$

Generalized consistency. In λ_i^+, a merge of function values, once applied to an argument, can step to a merge of expressions. In the previous example, for instance, one of the components in the resulting merge is $(2 + 1) : \mathsf{Int}$, which is an expression but not a value. This raises a challenge to the consistency definition employed in λ_i, which can only relate values (but not arbitrary expressions). Therefore we have to extend the definition of consistency in λ_i^+ to include such expressions. Intuitively, two expressions can be safely merged if their reduction result is the same, like $(2 + 1) : \mathsf{Int}$ or $3 \,,,\, (2 + 1)$. However, there are some difficulties regarding how to reason about expressions like the latter one. Consider two nonterminating programs, comparing them may never end. Instead, we model consistency with a syntactic definition, which is less powerful in the sense that it does not allow $3 \,,,\, (2 + 1)$. But such a definition is enough to accept the terms generated by parallel application, which keeps the syntactic equivalence among the components of merges.

Records. Together with BCD subtyping, single-field records and record types are added into λ_i^+. There is a distributivity rule in subtyping for records as well: $\{l : A\} \,\&\, \{l : B\} \leqslant \{l : A \,\&\, B\}$. A merge of several records can be used as a single record, as long as the records have the same field name. This is similar to function application where functions in one merge share one input. In reduction, we treat record projection like function application. That is to say, the parallel application relation not only applies functions in a merge in parallel but also projects records in a merge at the same time.

$$(\{l = \mathsf{True}\} \,,,\, \{l = 2\}).l \hookrightarrow \mathsf{True} \,,,\, 2$$

4 The λ_i calculus: Syntax, subtyping and typing

This section presents the type system of λ_i: a calculus with intersection types and a merge operator. This calculus is a variant of the original λ_i calculus (Oliveira *et al.*, 2016) (which is inspired by Dunfield (2014)'s calculus) with *fix points* and explicitly annotated lambdas instead of unannotated ones. Explicit annotations are necessary for the type-directed operational semantics of λ_i to preserve determinism. The TDOS can handle nonterminating programs, while some calculi using elaboration and coherence proofs (Bi *et al.*, 2018, 2019) do not support nonterminating programs. Dunfield (2014)'s calculus supports recursion, but it is incoherent.

4.1 Syntax

The syntax of λ_i is:

$$\begin{array}{lrcl}
\text{Types} & A, B & ::= & \mathsf{Int} \mid \mathsf{Top} \mid A \to B \mid A \,\&\, B \\
\text{Expressions} & e & ::= & x \mid i \mid \top \mid e : A \mid e_1\ e_2 \mid \lambda x.\, e : A \to B \mid e_1 \,,,\, e_2 \mid \mathsf{fix}\, x.\, e : A \\
\text{Values} & v & ::= & i \mid \top \mid \lambda x.\, e : A \to B \mid v_1 \,,,\, v_2 \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\text{Typing modes} & \Leftrightarrow & ::= & \Rightarrow \mid \Leftarrow
\end{array}$$

Types. Meta-variables A and B range over types. Two basic types are included: the integer type *Int* and the top type Top. Function types $A \to B$ and intersection types $A \,\&\, B$ can be used to construct compound types.

Expressions. Meta-variable e ranges over expressions. Expressions include some standard constructs: variables (x); integers (i); a canonical top value $\top$; annotated expressions ($e : A$); and application of a term e_1 to term e_2 (denoted by $e_1\ e_2$). Lambda abstractions ($\lambda x.e : A \to B$) must have a type annotation $A \to B$, meaning that the input type is A and the output type is B. The expression $e_1 \,,,\, e_2$ is the merge of expressions e_1 and e_2. Finally, fix points $\mathsf{fix}\, x.\, e : A$ (which also require a type annotation) model recursion.

Values, contexts, and typing modes. Meta-variable v ranges over values. Values include integers, the top value $\top$, lambda abstractions, and merges of values. Typing context Γ tracks bound variables (x) with their type A. $\Leftrightarrow$ stands for the mode of a bidirectional typing judgment: $\Rightarrow$ is the synthesis mode; $\Leftarrow$ is the checking mode. They differ on whether the type is an output (inferred) or an input (to be checked). The details of bidirectional typechecking will be discussed in Section 4.3.

4.2 Subtyping and disjointness

Subtyping. The subtyping rules are shown at the top of Figure 3. Here we follow the formalization by Davies & Pfenning (2000), except that we generalize rule S-TOP to allow any *top-like types* to be supertypes of any type. The original subtyping relation is known to be reflexive and transitive (Davies & Pfenning, 2000). We proved the reflexivity and transitivity of the extended subtyping relation as well.

Top-like types and arrow types. As suggested by its name, a top-like type is both a supertype and a subtype of Top. Besides Top, top-like types contain intersection types like Top & Top. In the middle of Figure 3 is its formal definition. Notably, rule TL-ARR allows arrow types to be top-like when their return types are top-like. This

$\boxed{A <: B}$ *(Subtyping)*

$$\frac{}{\mathsf{Int} <: \mathsf{Int}}\ \textsc{S-z} \qquad \frac{\rceil B \lceil}{A <: B}\ \textsc{S-top} \qquad \frac{A_1 <: A_3}{A_1 \,\&\, A_2 <: A_3}\ \textsc{S-andl1} \qquad \frac{A_2 <: A_3}{A_1 \,\&\, A_2 <: A_3}\ \textsc{S-andl2}$$

$$\frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}\ \textsc{S-arr} \qquad \frac{A_1 <: A_2 \qquad A_1 <: A_3}{A_1 <: A_2 \,\&\, A_3}\ \textsc{S-andr}$$

$\boxed{\rceil A \lceil}$ *(Top-Like Types)*

$$\frac{}{\rceil \mathsf{Top} \lceil}\ \textsc{TL-top} \qquad \frac{\rceil B \lceil}{\rceil A \to B \lceil}\ \textsc{TL-arr} \qquad \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil}\ \textsc{TL-and}$$

$\boxed{A *_a B}$ *(Algorithmic Disjointness)*

$$\frac{}{\mathsf{Top} *_a A}\ \textsc{D-topL} \qquad \frac{}{A *_a \mathsf{Top}}\ \textsc{D-topR} \qquad \frac{A_1 *_a B \qquad A_2 *_a B}{A_1 \,\&\, A_2 *_a B}\ \textsc{D-andL} \qquad \frac{A *_a B_1 \qquad A *_a B_2}{A *_a B_1 \,\&\, B_2}\ \textsc{D-andR}$$

$$\frac{}{\mathsf{Int} *_a A_1 \to A_2}\ \textsc{D-intArr} \qquad \frac{}{A_1 \to A_2 *_a \mathsf{Int}}\ \textsc{D-arrInt} \qquad \frac{A_2 *_a B_2}{A_1 \to A_2 *_a B_1 \to B_2}\ \textsc{D-arrArr}$$

Fig. 3. Subtyping, top-like types, and disjointness of λ_i.

enlargement of top-like types is inspired by the following rule in BCD-style subtyping (Barendregt *et al.*, 1983):

$$\frac{}{\mathsf{Top} <: \mathsf{Top} \to \mathsf{Top}}\ \textsc{BCD-TopArr}$$

We will come back to our motivation for allowing such top-like types in Section 4.3.

Disjointness. Section 3.4 presents the specification of disjointness. Such a specification is a liberal version of the original definition in λ_i. In our definition, A and B can be *top-like types*, which was forbidden in λ_i. An equivalent algorithmic definition of disjointness ($A *_a B$) is presented in the bottom of Figure 3, which is the same as the definition in the NeColus calculus (Bi *et al.*, 2018).

Lemma 4.1 (Disjointness properties)**.** *Disjointness satisfies:*

1. *$A * B$ if and only if $A *_a B$.*
2. *if $A * (B_1 \to C)$ then $A * (B_2 \to C)$.*
3. *if $A * B \,\&\, C$ then $A * B$ and $A * C$.*

4.3 Bidirectional typing

We use a bidirectional type system for λ_i to avoid a general subsumption rule, which causes ambiguity in the presence of a merge operator. A bidirectional type system has two kinds of typing judgments, each associated with one mode. The checking judgment

$\boxed{A \rhd B}$ *(Applicative Distributivity)*

$$\textsc{AD-arr}\quad \frac{}{A \to B \rhd A \to B} \qquad \textsc{AD-topArr}\quad \frac{}{\mathsf{Top} \rhd \mathsf{Top} \to \mathsf{Top}}$$

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

$$\textsc{Typ-top}\quad \frac{}{\Gamma \vdash \top \Rightarrow \mathsf{Top}} \qquad \textsc{Typ-lit}\quad \frac{}{\Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{Typ-var}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{Typ-abs}\quad \frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B}$$

$$\textsc{Typ-app}\quad \frac{\Gamma \vdash e_1 \Rightarrow C \qquad C \rhd A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\, e_2 \Rightarrow B} \qquad \textsc{Typ-merge}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad A * B}{\Gamma \vdash e_1 \,,,\, e_2 \Rightarrow A \,\&\, B}$$

$$\textsc{Typ-anno}\quad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A} \qquad \textsc{Typ-fix}\quad \frac{\Gamma, x : A \vdash e \Leftarrow A}{\Gamma \vdash \mathsf{fix}\, x.\, e : A \Rightarrow A}$$

$$\textsc{Typ-mergev}\quad \frac{\cdot \vdash v_1 \Rightarrow A \qquad \cdot \vdash v_2 \Rightarrow B \qquad v_1 \approx_{spec} v_2}{\Gamma \vdash v_1 \,,,\, v_2 \Rightarrow A \,\&\, B} \qquad \textsc{Typ-sub}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}$$

Fig. 4. Typing of λ_i.

$\Gamma \vdash e \Leftarrow A$ says that in the typing environment Γ, the expression e can be checked against type A, while the synthesis judgment $\Gamma \vdash e \Rightarrow A$ infers the type A from Γ and e. Unlike the original type system of λ_i (Figure 5), types have no well-formedness restriction, and expressions like $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ are allowed. This generalization is inspired by the NeColus calculus (Bi *et al.*, 2018), which is the first to introduce *unrestricted intersections* to a calculus which supports disjoint intersection types.

In Figure 4, most typing rules directly follow the bidirectional type system of the original λ_i, including the merge rule Typ-merge, where disjointness is used. When two expressions have disjoint types, any parts from each of them do not overlap in the types. Therefore, their merge does not introduce ambiguity. With this restriction, rule Typ-merge does not accept expressions like $1 \,,,\, 2$ or even $1 \,,,\, 1$. On the other hand, the novel rule Typ-mergev allows *consistent* values to be merged regardless of their types. It accepts $1 \,,,\, 1$ while still rejecting $1 \,,,\, 2$. The consistency specification used in rule Typ-mergev is given in Definition 3.1. It is for values only, and values are closed. Therefore, the premises should have an empty context (denoted by $\cdot$). As discussed in Section 3.4, together the two rules support the determinism and type preservation of the TDOS, and rule Typ-mergev does not need to be included in an implementation. The type system with the remaining rules is algorithmic. The rule Typ-fix is new and allows fix points. In addition, two rules are revised: rule Typ-abs and rule Typ-app. Since lambdas are now fully annotated in the current system, rule Typ-abs is changed from checking to synthesis mode. Next, we will discuss how rule Typ-app is generalized with the applicative distributivity relation.

Applicative distributivity and rule Typ-app. The top of Figure 4 shows the applicative distributivity relation, which relates a type with one of its arrow supertypes. Applicative distributivity is used in rule TYP-APP, where a term is expected to play the role of a function. Therefore, a term of type Top can be used as if it has type $\mathsf{Top} \to \mathsf{Top}$ and be applied to any terms. For example, $\top\ 1$ is allowed and it evaluates to $\top$.

Top-like types and merges of functions. We can finally come back to the motivation to allow arrow types in top-like types and depart from Dunfield's calculus. *If no arrow types are top-like*, two arrow types $A \to B$ and $C \to D$ are never disjoint in terms of Definition 3.1, as they have a common supertype $A \,\&\, C \to \mathsf{Top}$. Consequently, we can never create merges with more than one function, which is quite restrictive. For Dunfield this is not a problem because she does not have the disjointness restriction. So her calculus supports merges of any functions (but it is incoherent). In the original λ_i an ad hoc solution is proposed, by forcing the matter and employing the syntactic definition of top-like types in Figure 3 in disjointness, while keeping the standard rule $A <: \mathsf{Top}$ in subtyping. However this means that top-like function types are not supertypes of Top, which contradicts the intended meaning of a top-like type. In contrast, the approach we take in λ_i is to change the rule S-TOP in subtyping. Now $\mathsf{Top} <: (A \,\&\, C \to \mathsf{Top})$ is derivable and thus $A \,\&\, C \to \mathsf{Top}$ is genuinely a top-like type. In turn, this makes merges of multiple functions typeable without losing the intuition behind top-like types.

Checked subsumption. Unlike many calculi where there is a general subsumption rule that can apply anywhere, λ_i employs bidirectional type-checking, where subsumption is controlled. The subsumption (rule TYP-SUB) is in checking mode only. The checking mode is explicitly triggered by a type annotation, either via the rule TYP-ANNO, rule TYP-ABS or rule TYP-FIX. The annotation rule TYP-ANNO acts as explicit subsumption and assigns supertypes to expressions, provided a suitable type annotation. There is a strong motivation not to include a general (implicit) subsumption rule in calculi with disjoint intersection types. With an implicit subsumption rule, disjointness is insufficient to prevent some ambiguous terms, as shown in the following example.

$$\text{SUB}\ \dfrac{\text{MERGE}\ \dfrac{\text{SUB}\ \dfrac{\cdot \vdash 1 \Rightarrow \mathsf{Int} \qquad \mathsf{Int} <: \mathsf{Top}}{\cdot \vdash 1 \Rightarrow \mathsf{Top}} \qquad \cdot \vdash 2 \Rightarrow \mathsf{Int} \qquad \mathsf{Top} * \mathsf{Int}}{\cdot \vdash 1 ,, 2 \Rightarrow \mathsf{Top} \,\&\, \mathsf{Int}} \qquad \mathsf{Top} \,\&\, \mathsf{Int} <: \mathsf{Int} \,\&\, \mathsf{Int}}{\cdot \vdash 1 ,, 2 \Rightarrow \mathsf{Int} \,\&\, \mathsf{Int}}$$

Via the typical implicit subsumption, type Top is assigned to integer 1. Then 1 can be merged with 2 of type Int since their types are disjoint. At that time, the merged term $1 ,, 2$ has type $\mathsf{Top} \,\&\, \mathsf{Int}$, which is a subtype of $\mathsf{Int} \,\&\, \mathsf{Int}$. By applying the subsumption rule again, the ambiguous term $1 ,, 2$ finally bypasses the disjointness restriction, having type $\mathsf{Int} \,\&\, \mathsf{Int}$. However, note that with rule TYP-ANNO we can still type-check the term $(1 : \mathsf{Top}) ,, 2$, and reducing that term under the type *Int* can only unambiguously result in 2. The type annotation is key to prevent using the value 1 as an integer.

Typing properties. The bidirectional type-checking system has some properties that are important for the type soundness proof presented in Section 5. First, each term has only one synthesized type. Second, any well-typed term has a synthesized type, which is the principal type. Third, the type in a checking judgment can be replaced by a supertype.

Lemma 4.2 (Synthesis uniqueness). *If* $\Gamma \vdash e \Rightarrow A$ *and* $\Gamma \vdash e \Rightarrow B$ *, then* $A = B$.

Lemma 4.3 (Synthesis principality). *If* $\Gamma \vdash e \Leftarrow A$ *then there exists type B, s.t.* $\Gamma \vdash e \Rightarrow B$ *and* $B <: A$.

Lemma 4.4 (Checking subsumption). *If* $\Gamma \vdash e \Leftarrow A$ *and* $A <: B$*, then* $\Gamma \vdash e \Leftarrow B$.

4.4 Completeness with respect to the original type system

In this section, we discuss the relationship between the original λ_i (Oliveira *et al.*, 2016) and the new variant. To dmbiguate between the two calculi, we use λ_i '16 to denote the original calculus and λ_i for our variant. We prove that the type system of the new variant is at least as expressive as the λ_i '16 calculus[1]. The syntax of λ_i '16 (minus pairs and product types) is almost the same as λ_i, except that there are no fix points and the lambdas do not have any type annotations. Thus, lambdas can only be typed in checked mode. Figure 5 presents an excerpt of the type system. The type system has a type well-formedness definition and a slightly different disjointness relation compared to our variant of λ_i. Also note that the rule for the merge of values (rule TYP-MERGEV) is absent because the disjointness restriction in well-formedness prevents duplicated values.

Some details need to be explained before presenting the completeness theorem. First, subtyping in our variant of λ_i is stronger due to top-like types. Second, top-like types are disjoint to any type in our variant, while the disjointness in the original λ_i '16 is restricted to types that are not top-like. We extended the bidirectional type system of the original λ_i '16 with recursion and designed an elaboration from the extended system to λ_i. We proved a theorem that shows the type system of λ_i can type check any well-typed terms in λ_i '16, with type annotations inserted based on the typing derivation:

Theorem 4.1 (Completeness of typing with respect to the extended original λ_i). *If* $\Gamma \models E \Leftrightarrow A \hookrightarrow e$*, then* $\Gamma \vdash e \Leftrightarrow A$.

The result means that the type system of λ_i '16 (or any type system equivalent to it) can be used as a surface language where many of the explicit annotations of λ_i are inferred automatically. That is to say, the λ_i '16 calculus can be translated into λ_i without loss of expressivity or flexibility. Moreover, the extension of fix points further shows that some type inference with recursion is feasible.

5 A type-directed operational semantics for λ_i

This section introduces the type-directed operational semantics for λ_i. The operational semantics uses type information arising from type annotations to guide the reduction

[1] Note that the original λ_i includes pairs and product types. In the Coq formalization we have a variant with pairs and product types as well. It has all the previous properties proved in this section. For simplicity and consistency of presentation, we use the variant without pairs and product types in the paper.

$\boxed{\Gamma \models A}$ *(Type Well-formedness)*

$$\textsc{Wf-top}\ \frac{}{\Gamma \models \mathsf{Top}} \qquad \textsc{Wf-int}\ \frac{}{\Gamma \models \mathsf{Int}} \qquad \textsc{Wf-arr}\ \frac{\Gamma \models A \qquad \Gamma \models B}{\Gamma \models A \to B} \qquad \textsc{Wf-and}\ \frac{\Gamma \models A \qquad \Gamma \models B \qquad A * B}{\Gamma \models A \,\&\, B}$$

$\boxed{\Gamma \models E \Leftrightarrow A \hookrightarrow e}$ *(Bidirectional Typing)*

$$\textsc{IBTyp-top}\ \frac{}{\Gamma \models \top \Rightarrow \mathsf{Top} \hookrightarrow \top} \qquad \textsc{IBTyp-lit}\ \frac{}{\Gamma \models i \Rightarrow \mathsf{Int} \hookrightarrow i} \qquad \textsc{IBTyp-var}\ \frac{x : A \in \Gamma}{\Gamma \models x \Rightarrow A \hookrightarrow x}$$

$$\textsc{IBTyp-app}\ \frac{\begin{array}{c}\Gamma \models E_1 \Rightarrow A \to B \hookrightarrow e_1 \\ \Gamma \models E_2 \Leftarrow A \hookrightarrow e_2\end{array}}{\Gamma \models E_1\, E_2 \Rightarrow B \hookrightarrow e_1\, e_2} \qquad \textsc{IBTyp-merge}\ \frac{\begin{array}{c}\Gamma \models E_1 \Rightarrow A \hookrightarrow e_1 \\ \Gamma \models E_2 \Rightarrow B \hookrightarrow e_2 \qquad A * B\end{array}}{\Gamma \models E_1 \,,,\, E_2 \Rightarrow A \,\&\, B \hookrightarrow e_1 \,,,\, e_2}$$

$$\textsc{IBTyp-anno}\ \frac{\Gamma \models E \Leftarrow A \hookrightarrow e}{\Gamma \models E : A \Rightarrow A \hookrightarrow e : A} \qquad \textsc{IBTyp-fix}\ \frac{\Gamma \models A \qquad \Gamma, x : A \models E \Leftarrow A \hookrightarrow e}{\Gamma \models \mathsf{fix}\, x.\, E \Leftarrow A \hookrightarrow \mathsf{fix}\, x.\, e : A}$$

$$\textsc{IBTyp-lam}\ \frac{\Gamma \models A \qquad \Gamma, x : A \models E \Leftarrow B \hookrightarrow e}{\Gamma \models \lambda x.\, E \Leftarrow A \to B \hookrightarrow (\lambda x.\, e : A \to B)} \qquad \textsc{IBTyp-sub}\ \frac{\Gamma \models E \Rightarrow A \hookrightarrow e \qquad A <: B}{\Gamma \models E \Leftarrow B \hookrightarrow e}$$

Fig. 5. The type system of the original λ_i (extended with fix points). Three rules that relate with product types are ignored.

process. In particular, a new relation called *typed reduction* is used to further reduce values based on the contextual type information, forcing the value to match the type structure. We show two important properties for λ_i: *determinism of reduction* and *type soundness*. That is to say, there is only one way to reduce an expression according to the small-step relation, and the process preserves types and never gets stuck.

5.1 Typed reduction of values

To account for the type information during reduction, λ_i uses an auxiliary reduction relation called *typed reduction* for reducing values under a certain type. Typed reduction $v \hookrightarrow_A v'$ reduces the value v under type A, producing a value v' that has type A. It arises when given a value v of some type, where A is a supertype of the type of v, and v needs to be converted to a value compatible with the supertype A. Typed reduction ensures that values and types have a strong correspondence. If a value is well-typed, its principal type can be told directly by looking at its syntactic form. Typed reduction can be viewed as a relation that gives a runtime interpretation to subtyping, and the rules of typed reduction are aligned in a one-by-one correspondence with subtyping. While subtyping states what kind of conversions are valid at the type level, typed reduction gives an operational meaning for such conversions on values.

Figure 6 shows the typed reduction relation. Rule TR-TOP expresses the fact that a top-like type is the supertype of any type, which means that any value can be reduced under it. The top-like type is restricted to be *ordinary* (Davies & Pfenning, 2000),

$\boxed{v \hookrightarrow_A v'}$ *(Typed Reduction)*

$$\textsc{TR-lit}\ \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \qquad \textsc{TR-top}\ \frac{A\ \mathsf{Ordinary} \qquad \rceil A \lceil}{v \hookrightarrow_A \top} \qquad \textsc{TR-arrow}\ \frac{\neg \rceil A_2 \lceil \qquad A_1 <: B_1 \qquad B_2 <: A_2}{\lambda x.\, e : B_1 \to B_2 \hookrightarrow_{(A_1 \to A_2)} \lambda x.\, e : B_1 \to A_2}$$

$$\textsc{TR-mergevl}\ \frac{v_1 \hookrightarrow_A v_1' \qquad A\ \mathsf{Ordinary}}{v_1 ,, v_2 \hookrightarrow_A v_1'} \qquad \textsc{TR-mergevr}\ \frac{v_2 \hookrightarrow_A v_2' \qquad A\ \mathsf{Ordinary}}{v_1 ,, v_2 \hookrightarrow_A v_2'} \qquad \textsc{TR-and}\ \frac{v \hookrightarrow_A v_1 \qquad v \hookrightarrow_B v_2}{v \hookrightarrow_{A \,\&\, B} v_1 ,, v_2}$$

Fig. 6. Typed reduction of λ_i.

to avoid overlapping with the rule TR-AND. Ordinary types are all types that are not intersections:

$\boxed{A\ \mathsf{Ordinary}}$ *(Ordinary Types)*

$$\textsc{O-top}\ \frac{}{\mathsf{Top}\ \mathsf{Ordinary}} \qquad \textsc{O-int}\ \frac{}{\mathsf{Int}\ \mathsf{Ordinary}} \qquad \textsc{O-arrow}\ \frac{}{A \to B\ \mathsf{Ordinary}}$$

The rule TR-TOP indicates that under such a type, any value reduces to the top value $\top$. Recall that the top-like definition in Figure 3 includes arrow types whose return type is top-like, thus the rule TR-TOP covers values with such top-like arrow types as well. Rule TR-LIT expresses that an integer value reduced under the supertype Int is just the integer value itself. Rule TR-ARROW states that a lambda value $\lambda x.\, e : A \to B$, under a *non-top-like type* $C \to D$, evaluates to $\lambda x.\, e : A \to D$ if $C <: A$ and $B <: D$. The restriction that $C \to D$ is not top-like avoids overlapping with rule TR-TOP. Importantly rule TR-ARROW changes the return type of lambda abstractions, and keeps the input type, since it is needed at runtime (by rule STEP-BETA which is discussed in Section 5.3).

Intersections and merges. In the remaining rules, we first decompose intersections. Then we only need to consider ordinary types. We take care of the value by going through every merge, until both the value and type are in a basic form. Rule TR-MERGEVL and rule TR-MERGEVR are a pair of rules for reducing merges under an ordinary type. Since the type is not an intersection, the result contains no merge. Usually, we need to select between the left part and right part of a merge according to the type. The values of disjoint types do not overlap on non-top-like types. For example, $1 ,, (\lambda x.\, x : \mathsf{Int} \to \mathsf{Int}) \hookrightarrow_{\mathsf{Int}} 1$ selects the left part. For top-like types, no matter which rule is applied, the reduction result is determined by the type only, as the rule TR-TOP suggests.

Rule TR-AND is the rule that deals with intersection types. It says that if a value v can be reduced to v_1 under type A and can be reduced to v_2 under type B, then its reduction result under type $A \,\&\, B$ is the merge of two results $v_1 ,, v_2$. Note that this rule may *duplicate values*. For example, $1 \hookrightarrow_{\mathsf{Int} \,\&\, \mathsf{Int}} 1 ,, 1$. Such duplication requires special care, since the merge violates disjointness. The specially designed typing rule (rule TYP-MERGEV) uses the notion of consistency (Definition 3.2) instead of disjointness to type-check a merge of two values. Note also that such duplication implies that sometimes it is possible to use either rule TR-MERGEVL or rule TR-MERGEVR to reduce a value. For example, 1 ,,

$1 \hookrightarrow_{\mathsf{Int}} 1$. The consistency restriction in rule Typ-mergev ensures that no matter which rule is applied in such a case, the result is the same.

Example. A larger example to demonstrate how typed reduction works is

$$\begin{aligned} & (\lambda x.\,(x \,,,\, \text{‘}c\text{’}) : \mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Char}) \,,,\, (\lambda x.\,x : \mathsf{Bool} \to \mathsf{Bool}) \,,,\, 1 \\ \hookrightarrow_{\mathsf{Int}\,\&\,(\mathsf{Int}\to\mathsf{Int})} \; & 1 \,,,\, (\lambda x.\,(x \,,,\, \text{‘}c\text{’}) : \mathsf{Int} \to \mathsf{Int}) \end{aligned}$$

The initial value is the merge of two lambda abstractions and an integer. The target type is $\mathsf{Int} \,\&\, (\mathsf{Int} \to \mathsf{Int})$. Because the target type is an intersection, typed reduction first employs rule TR-and to decompose the intersection into Int and $\mathsf{Int} \to \mathsf{Int}$. Under type Int the value reduces to 1, and under type $\mathsf{Int} \to \mathsf{Int}$ it will reduce to $\lambda x.\,x \,,,\, \text{‘}c\text{’} : \mathsf{Int} \to \mathsf{Int}$. Therefore, we obtain the merge $1 \,,,\, (\lambda x.\,x \,,,\, \text{‘}c\text{’} : \mathsf{Int} \to \mathsf{Int})$ with type $\mathsf{Int} \,\&\, (\mathsf{Int} \to \mathsf{Int})$.

Basic properties of typed reduction. Some properties of typed reduction can be proved directly by induction on the typed reduction derivation. First, when typed reduction is under a top-like type, the result only depends on the type. Second, typed reduction produces the same result whenever it is done directly or indirectly. Third, if a well-typed value can be type-reduced by some type, its synthesized type must be a subtype of that type. The three properties are formally stated next:

Lemma 5.1 (Top-like typed reduction). *If* $\rceil A \lceil$, $v_1 \hookrightarrow_A v_1'$ *, and* $v_2 \hookrightarrow_A v_2'$ *then* $v_1' = v_2'$.

Lemma 5.2 (Typed reduction transitivity). *If* $v \hookrightarrow_A v_1$*, and* $v_1 \hookrightarrow_B v_2$*, then* $v \hookrightarrow_B v_2$.

Lemma 5.3 (Subtyping preservation). *If* $v \hookrightarrow_A v'$ *and* $\cdot \vdash v \Rightarrow B$*, then* $B <: A$.

Note that Lemma 5.3 relates typed reduction and subtyping.

5.2 Consistency, determinism and type soundness of typed reduction

Consistent values, as specified in Definition 3.2, introduce no ambiguity in typed reduction. If two consistent values both can reduce under a type, they should produce the same result. The *consistency* restriction ensures that duplicated values in a merge type-check, but it still rejects merges with different values of the same type. A value of a top-like type is consistent with any other value. It only type-reduces under top-like types, which leads to a fixed result decided by the type.

Relating disjointness and consistency. Assuming that the synthesized types of two values are disjoint, from Lemma 5.3, we can conclude that when the two values both reduce under a type, that type must be a common supertype of their principal types, which is known to be top-like. Furthermore, Lemma 5.1 implies that their reduction results are always the same under such top-like types, so they are consistent. The above discussion concludes that values with disjoint types evaluate to the same result under the same type, i.e. they are consistent. This is captured by the following lemma:

Lemma 5.4 (Consistency of disjoint values). *If* $A * B$, $\cdot \vdash v_1 \Rightarrow A$*, and* $\cdot \vdash v_2 \Rightarrow B$*, then* $v_1 \approx_{spec} v_2$.

$\boxed{A \ll B}$ *(Runtime Subtyping)*

$$\textsc{RSub-z}\ \frac{}{A \ll A} \qquad \textsc{RSub-arr}\ \frac{B_1 <: A_1 \qquad A_2 \ll B_2}{A_1 \to A_2 \ll B_1 \to B_2} \qquad \textsc{RSub-and}\ \frac{A_1 \ll B_1 \qquad A_2 \ll B_2}{A_1 \,\&\, A_2 \ll B_1 \,\&\, B_2} \qquad \textsc{RSub-top}\ \frac{\rceil A \lceil}{\mathsf{Top} \ll A}$$

Fig. 7. Runtime subtyping of λ_i.

Determinism of typed reduction. The merge construct makes it hard to design a deterministic operational semantics. Disjointness and consistency restrictions prevent merges like $1 ,, 2$, and bring the possibility to deal with merges based on types. Typed reduction takes a well-typed value, which, if it is a merge, must be consistent (according to Lemma 5.4). When the two typed reduction rules for merges (rule TR-MERGEVL and rule TR-MERGEVR) overlap, no matter which one is chosen, either value reduces to the same result due to consistency. Indeed our typed reduction relation always produces a unique result for any legal combination of the input value and type. This serves as a foundation for the determinism of the operational semantics.

Lemma 5.5 (Determinism of typed reduction)**.** *For every well-typed v (that is there is some type B such that $\cdot \vdash v \Rightarrow B$), if $v \hookrightarrow_A v_1$ and $v \hookrightarrow_A v_2$ then $v_1 = v_2$.*

Runtime subtyping. While most typed reduction rules produce values of the reduction type (in synthesis mode), two rules are more relaxed. Rule TR-TOP offers $\top$ for any top-like types. Rule TR-ARROW keeps the original input annotation in the reduced lambda:

$$(\lambda x.\, x ,, 2 : \mathsf{Char} \to \mathsf{Char} \,\&\, \mathsf{Int}) \hookrightarrow_{(\mathsf{Char} \,\&\, \mathsf{Int} \to \mathsf{Char})} \lambda x.\, x ,, 2 : \mathsf{Char} \to \mathsf{Char}$$

Precisely speaking, the synthesized type of the result in typed reduction is a *runtime subtype* of the reduction type. Defined in Figure 7, runtime subtyping is a restricted form of subtyping. Roughly, runtime subtyping only allows subtyping in contravariant positions except for top-like types.

Runtime subtyping is introduced because we need to find a middle point between equality and subtyping to describe how typed reduction preserves the reduction type (Lemma 5.7). If A is the reduction type and B is the type of the output value in the typed reduction relation, we cannot simply say that $B = A$. But knowing only that $B <: A$ is not enough. It does not even prevent directly using the input value as the result. Runtime subtyping ensures that the reduction result behaves like a term of the reduction type, and it keeps transitivity as well. Thus, after multiple steps of reduction, the ultimate result still has a runtime subtype in terms of the type of the initial expression. Therefore, the preservation property of λ_i is safely relaxed, to allow the expression type to become more and more specific during reduction.

Type soundness of typed reduction. Via the transitivity lemma (Lemma 5.2) and the determinism lemma (Lemma 5.5), we obtain the following property: any reduction results of the given value are consistent.

Lemma 5.6 (Consistency after typed reduction)**.** *If v is well typed , and $v \hookrightarrow_A v_1$, and $v \hookrightarrow_B v_2$ then $v_1 \approx_{spec} v_2$.*

The lemma shows that the reduction result of rule TR-AND is always made of consistent values, which is needed in type preservation via the typing rule TYP-MERGEV. Then, a (generalized) type preservation lemma on typed reduction can be proved.

Lemma 5.7 (Preservation of typed reduction)**.** *If* $\cdot \vdash v \Leftarrow A$ *and* $v \hookrightarrow_A v'$, *then* $\exists B$, $\cdot \vdash v' \Rightarrow B$ *and* $B \ll A$.

In general, this lemma shows that typed reduction produces well-typed values: it shows that if a value is checked by type A and it can be type reduced by A, then the reduced value is always well typed, and its synthesized type B is a *runtime subtype* of A. What is more, typed reduction is guaranteed to progress for a given value and a type it can be checked against. That is to say, from a well-typed value, we can derive the existence of a typed reduction judgement and the well typedness of the reduction result.

Lemma 5.8 (Progress of typed reduction)**.** *If* $\cdot \vdash v \Leftarrow A$, *then* $\exists v'$, $v \hookrightarrow_A v'$.

Fewer checks on typed reduction. In rule TR-ARROW (in Figure 6), the premise $A_1 <: B_1$ is redundant for reduction. Since we only care about well-typed terms being reduced, such a check has already been guaranteed by typing. Therefore, an actual implementation could omit that check. The reason why we keep the premise is that typed reduction plays another role in our metatheory: it allows us to define consistency. Consistency is defined for any (untyped) values, and the extra check there tightens up the definition of consistency. With the premise, typed reduction directly implies a subtyping relation between the type of the reduced value and the reduction type. (See Lemma 5.3: If $v \hookrightarrow_A v'$, and $\cdot \vdash v \Rightarrow B$, then $B <: A$.)

One could wonder if this property is unnecessary because it may be derived by type preservation of reduction. Note that whenever typed reduction is called in a reduction rule, the subtyping relation can be obtained from the typing derivation of the reduced term. For example, reducing $v : A$ will type-reduce v under A. If $v : A$ is well typed, then we could in principle prove that the type of v is a subtype of A. Unfortunately, the above proof is hard to attain in practice. Because type preservation depends on consistency, and consistency is defined by typed reduction. Once the subtyping property relies on type preservation, there is a cyclic dependency between the properties.

5.3 Reduction

The reduction rules are presented in Figure 8. Recall that rule TYP-APP is generalized using applicative distributivity. Correspondingly, the top value consumes every input it meets using rule STEP-TOP. Pierce & Steffen (1997) employ a similar rule in a calculus with higher-order subtyping. Rule STEP-BETA and rule STEP-ANNOV are the two rules relying on typed reduction judgments. Rule STEP-BETA says that a lambda value $\lambda x.\, e : A \to B$ applied to value v reduces by replacing the bound variable x in e by v'. Importantly, v' is obtained by type-reducing v under type A. In other words, in rule STEP-BETA further (typed) reduction may be necessary on the argument depending on its type. This is unlike many other calculi where values are in a final form and no further reduction is needed

$$\boxed{e \hookrightarrow e'} \qquad \qquad \textit{(Reduction)}$$

$$\textsc{Step-top}\ \frac{}{\top\, v \hookrightarrow \top} \qquad \textsc{Step-beta}\ \frac{v \hookrightarrow_A v'}{(\lambda x.\, e : A \to B)\ v \hookrightarrow (e[x \mapsto v']) : B} \qquad \textsc{Step-annov}\ \frac{v \hookrightarrow_A v'}{v : A \hookrightarrow v'} \qquad \textsc{Step-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\, e_2 \hookrightarrow e_1'\, e_2}$$

$$\textsc{Step-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\, e_2 \hookrightarrow v_1\, e_2'} \qquad \textsc{Step-mergel}\ \frac{e_1 \hookrightarrow e_1'}{e_1 \,,,\, e_2 \hookrightarrow e_1' \,,,\, e_2} \qquad \textsc{Step-merger}\ \frac{e_2 \hookrightarrow e_2'}{v_1 \,,,\, e_2 \hookrightarrow v_1 \,,,\, e_2'} \qquad \textsc{Step-anno}\ \frac{e \hookrightarrow e'}{e : A \hookrightarrow e' : A}$$

$$\textsc{Step-fix}\ \frac{}{(\mathsf{fix}\, x.\, e : A) \hookrightarrow (e[x \mapsto (\mathsf{fix}\, x.\, e : A)]) : A}$$

Fig. 8. Call-by-value reduction of λ_i.

before substitution. The rule STEP-ANNOV says that an annotated $v : A$ can be reduced to v' if v type-reduces to v' under type A.

Metatheory of reduction. When designing the operational semantics of λ_i, we want it to have two properties: *determinism of reduction* and *type soundness*. That is to say, there is only one way to reduce an expression according to the small-step relation, and the process preserves types and never gets stuck. Similar lemmas on typed reduction were already presented, which are necessary for proving the following theorems, mainly in cases related to rule STEP-ANNOV and rule STEP-BETA.

Theorem 5.1 (Determinism of $\hookrightarrow$). *If $\cdot \vdash e \Leftarrow A$, $e \hookrightarrow e_1$, $e \hookrightarrow e_2$, then $e_1 = e_2$.*

The preservation theorem states that during reduction, the program is always well typed, and the reduced expression can be checked against the original type.

Theorem 5.2 (Type preservation of $\hookrightarrow$). *If $\cdot \vdash e \Leftrightarrow A$, and $e \hookrightarrow e'$ then $\cdot \vdash e' \Leftarrow A$.*

This theorem is a corollary of the following lemma:

Lemma 5.9 (Generalized type preservation of $\hookrightarrow$). *If $\cdot \vdash e \Leftrightarrow A$, and $e \hookrightarrow e'$ then $\exists B$, $\cdot \vdash e' \Leftrightarrow B$ and $B \ll A$.*

The lemma has a similar structure to Lemma 5.7: the type of the reduced result is a runtime subtype (Figure 7) of the target type. Note that $\Leftrightarrow$ in this and the following lemmas is a meta-variable for typing mode. It means both checking and synthesis mode work for it, as long as the conclusion and the premise have the same mode. To prove Lemma 5.9, the substitution lemma has to be adapted. The substituted term is allowed to have a runtime subtype of the expected type. The type of the result, accordingly, is a subtype of the initial type. For example, a lambda of type $\mathsf{Int} \to \mathsf{Int}$ can be used when a term of $\mathsf{Int} \,\&\, \mathsf{Char} \to \mathsf{Int}$ is expected. It can be viewed as a combination of type narrowing via runtime subtyping and the conventional substitution lemma.

Lemma 5.10 (Substitution preserves types). *For any expression e, if $\Gamma_1, x : B, \Gamma_2 \vdash e_1 \Leftrightarrow A$ and $\Gamma_2 \vdash e_2 \Rightarrow B'$, $B' \ll B$, then $\Gamma_1, \Gamma_2 \vdash e_1[x \mapsto e_2] \Leftrightarrow A'$ and $A' \ll A$.*

Finally, the progress theorem promises that reduction never gets stuck. Its proof relies on the progress lemma of typed reduction.

$$
\begin{array}{rcl}
|\, i \,| & = & i \\
|\, \top \,| & = & \top \\
|\, \lambda x.\, e : A \to B \,| & = & \lambda x.\, |\, e \,| \\
|\, \mathsf{fix}\, x.\, e : A \,| & = & \mathsf{fix}\, x.\, |\, e \,| \\
|\, e : A \,| & = & |\, e \,| \\
|\, e_1\, e_2 \,| & = & |\, e_1 \,|\, |\, e_2 \,| \\
|\, e_1 \,,,\, e_2 \,| & = & |\, e_1 \,| \,,,\, |\, e_2 \,|
\end{array}
$$

Fig. 9. Type erasure of λ_i.

Theorem 5.3 (Progress of $\hookrightarrow$). *If* $\cdot \vdash e \Leftarrow A$*, then* e *is a value or* $\exists e'$*,* $e \hookrightarrow e'$*.*

5.4 Soundness with respect to Dunfield's operational semantics

Dunfield's nondeterministic operational semantics motivates our TDOS. Here, we show the soundness of the operational semantics of λ_i with respect to a slightly extended version of Dunfield's semantics. The need for extending Dunfield's original semantics is mostly due to the generalization of the rule S-TOP in subtyping. In the conference version of this paper (Huang & Oliveira, 2020) we also discuss a variant of λ_i (which uses the original subtyping) and show that such a variant requires no changes to Dunfield's semantics.

Dunfield's original reduction rules are presented in Fig 1. We extend her operational semantics with the following two rules.

$\boxed{E \rightsquigarrow E'}$ *(The Extension of Dunfield's Calculus)*

$$
\textsc{DStep-top}\ \frac{}{V \rightsquigarrow \top}
\qquad\qquad
\textsc{DStep-topArr}\ \frac{}{\top\, V \rightsquigarrow \top}
$$

Rule DSTEP-TOPARR states that the value $\top$ can be used as a lambda which returns $\top$, suggested by the newly added top-like types for arrow types returning *Top*. Rule DSTEP-TOP states that any value can be reduced to $\top$, corresponding to $A <: \mathsf{Top}$. Dunfield avoids having a rule DSTEP-TOP by performing a simplifying elaboration step in advance:

$$
\frac{}{\Gamma \vdash V : \mathsf{Top} \hookrightarrow \top}\ \textsc{Dunfield-Typing-T}
$$

With such a rule, values of type Top are directly translated into $\top$ and do not need any further reduction in the target language. We do not have such an elaboration step. Instead we extend the original semantics with the two rules above.

Type erasure. Differently from Dunfield's calculus, λ_i uses type annotations in its syntax to obtain a direct operational semantics. $|\, e \,|$ erases annotations in term e. By erasing all annotations, terms in λ_i can be converted to terms in Dunfield's calculus (and also the original λ_i). The annotation erasure function is defined in Figure 9. Note that for every value v in λ_i, $|\, v \,|$ is a value as well.

Soundness. Given Dunfield's extended semantics, we can show a theorem that each step in the TDOS of λ_i corresponds to zero, one, or multiple steps in Dunfield's semantics.

Theorem 5.4 (Soundness of $\hookrightarrow$ with respect to Dunfield's semantics). *If $e \hookrightarrow e'$, then* $|\, e \,| \rightsquigarrow^* |\, e' \,|$.

A necessary auxiliary lemma for this theorem is the soundness of typed reduction.

Lemma 5.11 (Soundness of typed reduction with respect to Dunfield's semantics). *If* $v \hookrightarrow_A v'$, *then* $|\, v \,| \rightsquigarrow^* |\, v' \,|$.

This lemma shows that although the type information guides the reduction of values, it does not add additional behavior to values. For example, a merge can step to its left part (or the right part) with rule TR-MERGEVL (or rule TR-MERGEVR), corresponding to rule DSTEP-UNMERGEL (or rule DSTEP-UNMERGER). Rule TR-AND can be understood as a combination of splitting (rule DSTEP-SPLIT $V \rightsquigarrow V \,,, V$) and further reduction on each component separately.

6 A modular and algorithmic formulation of BCD subtyping

The formalization of λ_i^+ in Section 7 is an extension of λ_i. At the type level, the main addition of λ_i^+ over λ_i is a more powerful subtyping relation based on BCD subtyping (Barendregt *et al*., 1983). In this section, we first revisit BCD subtyping and propose a new modular and algorithmic formulation of BCD subtyping. This new algorithmic formulation of BCD subtyping is important for the design of the typed reduction relation for λ_i^+. The most interesting feature in BCD subtyping is its distributivity rule between intersection and function types. However, such a rule introduces complications, and designing sound and complete algorithms is tricky. In particular, in previous work (Bi *et al*., 2018; Pierce, 1989; Bessai *et al*., 2016, 2019; Siek, 2019), the distributivity rule leads to non-modular algorithmic formulations where many standard subtyping rules have to be changed due to distributivity. Furthermore, the metatheory of BCD subtyping is challenging.

We propose a novel modular and algorithmic BCD formulation. The key idea is to use the novel notion of *splittable types*, which are types that can be split into an intersection of two simpler types. We show basic properties of our formulation, including transitivity and inversion lemmas, and conclude that it is sound and complete with respect to the declarative BCD subtyping. Of particular interest is our transitivity proof. This proof is remarkably simple in comparison with other proofs in the literature due to a semantic characterization of types using splittable and ordinary types (Davies & Pfenning, 2000), which is used as the inductive argument for transitivity.

6.1 BCD subtyping

The BCD subtyping relation supports intersection types and allows some forms of distributivity. The original BCD formulation is shown in Figure 10. Most notably, BCD subtyping supports distributivity of intersections over function types using the rule OS-DISTARR. This rule says that an intersection of two function types $A \to B$ and $A \to C$ is a subtype of a function type $A \to B \,\&\, C$.

The rule OS-TOPARR is also interesting: in combination with the transitivity rule, it essentially allows Top to be a subtype of any function type returning Top (recall also

$\boxed{A \leqslant B}$ *(Original BCD Declarative Subtyping)*

$$\textsc{OS-refl}\ \frac{}{A \leqslant A} \qquad \textsc{OS-trans}\ \frac{A \leqslant B \qquad B \leqslant C}{A \leqslant C} \qquad \textsc{OS-top}\ \frac{}{A \leqslant \mathsf{Top}} \qquad \textsc{OS-topArr}\ \frac{}{\mathsf{Top} \leqslant \mathsf{Top} \to \mathsf{Top}}$$

$$\textsc{OS-arr}\ \frac{B \leqslant A \qquad C \leqslant D}{A \to C \leqslant B \to D} \qquad \textsc{OS-and}\ \frac{A \leqslant B \qquad A \leqslant C}{A \leqslant B \,\&\, C} \qquad \textsc{OS-andl}\ \frac{}{A \,\&\, B \leqslant A} \qquad \textsc{OS-andr}\ \frac{}{A \,\&\, B \leqslant B}$$

$$\textsc{OS-distArr}\ \frac{}{(A \to B) \,\&\, (A \to C) \leqslant A \to B \,\&\, C}$$

Fig. 10. Declarative BCD subtyping.

the discussion in Section 4.2). In fact, the relation $\mathsf{Top} <: \mathsf{Top} \to \mathsf{Top}$ is equivalent to $\mathsf{Top} <: A \to \mathsf{Top}$, since $\mathsf{Top} \to \mathsf{Top}$ is the lower bound of $A \to \mathsf{Top}$ for any type A. With BCD subtyping, we can further justify the property $\mathsf{Top} <: A \to \mathsf{Top}$ by the distributivity rule (extended to multiple components):

$$(A \to B_1) \,\&\, (A \to B_2) \,\&\, \ldots \,\&\, (A \to B_n) \leqslant A \to B_1 \,\&\, B_2 \,\&\, \ldots \,\&\, B_n$$

The Top type is commonly treated as an intersection of zero types. Therefore, when $n = 0$, the above subtyping judgment becomes $\mathsf{Top} <: A \to \mathsf{Top}$. On the other hand, from the coercive subtyping point of view, a simple coercion that validates the relation can be a constant function that returns $\lambda x.\, \top : A \to \mathsf{Top}$.

The reflexivity and transitivity rules are common elements for declarative systems. In this particular system, the transitivity rule is hard to eliminate, mainly due to the existence of rule OS-distArr.

6.2 A simple and modular formulation of BCD with splittable types

In order to obtain an algorithm for the BCD subtyping, the transitivity rule must be eliminated. As a step toward transitivity elimination, we treat any type A that is equivalent to an intersection type directly as an equivalent intersection type $B \,\&\, C$. If such treatment is possible, we call A *splittable*; otherwise, A is *ordinary*.

Ordinary types. Ordinary types (Davies & Pfenning, 2000) have been used in the past to define algorithmic formulations of subtyping with intersection types (but without distributivity). At the top of Figure 11, we present the definition of ordinary types for our formulation. Traditionally, an ordinary type is any type that is not an intersection of two other types. However, in this part of the paper, this distinction is more fine-grained, since some function types may not be ordinary. For a function type to be ordinary its output type must be ordinary as well.

Splittable types. The splittable relation, also shown in Figure 11, can be viewed as taking an input type A, and returning two types B and C, such that A is equivalent to $B \,\&\, C$, i.e. $A <: B \,\&\, C \ \wedge\ B \,\&\, C <: A$. Rule Sp-and splits an intersection type directly. Rule Sp-arrow splits a function type when its return type is splittable. The reasoning

$$\boxed{A\ \mathsf{Ordinary}^{+}} \qquad \textit{(Ordinary Types)}$$

$$\frac{}{\mathsf{Top\ Ordinary}^{+}}\ \textsc{O-bcd-top} \qquad \frac{}{\mathsf{Int\ Ordinary}^{+}}\ \textsc{O-bcd-int} \qquad \frac{B\ \mathsf{Ordinary}^{+}}{A \to B\ \mathsf{Ordinary}^{+}}\ \textsc{O-bcd-arrow}$$

$$\boxed{B \lhd A \rhd C} \qquad \textit{(Splittable Types)}$$

$$\frac{}{A \lhd A \,\&\, B \rhd B}\ \textsc{Sp-and} \qquad \frac{C \lhd B \rhd D}{A \to C \lhd A \to B \rhd A \to D}\ \textsc{Sp-arrow}$$

$$\boxed{A <: B} \qquad \textit{(Modular BCD Subtyping)}$$

$$\frac{}{\mathsf{Int} <: \mathsf{Int}}\ \textsc{S-bcd-int} \qquad \frac{B\ \mathsf{Ordinary}^{+} \qquad \rceil B \lceil}{A <: B}\ \textsc{S-bcd-top} \qquad \frac{D\ \mathsf{Ordinary}^{+} \qquad B <: A \qquad C <: D}{A \to C <: B \to D}\ \textsc{S-bcd-arr}$$

$$\frac{C\ \mathsf{Ordinary}^{+} \qquad A <: C}{A \,\&\, B <: C}\ \textsc{S-bcd-andl} \qquad \frac{C\ \mathsf{Ordinary}^{+} \qquad B <: C}{A \,\&\, B <: C}\ \textsc{S-bcd-andr}$$

$$\frac{B \lhd D \rhd C \qquad A <: B \qquad A <: C}{A <: D}\ \textsc{S-bcd-and}$$

Fig. 11. Algorithmic and modular subtyping.

for rule Sp-arrow is that both $A \to B \,\&\, C <: (A \to B) \,\&\, (A \to C)$ and $(A \to B) \,\&\, (A \to C) <: A \to B \,\&\, C$ are derivable in declarative BCD subtyping. The following lemma provides some justification for type splitting. It is proven by a routine induction on the splittable premise.

Lemma 6.1 (Splittable subtypes)**.** *If $B_1 \lhd B \rhd B_2$ and $A \leqslant B_1$ and $A \leqslant B_2$, then $A \leqslant B$.*

Three important properties related to ordinary and splittable types are:

Lemma 6.2 (Ordinary types do not split)**.** *For any ordinary type A, A is not splittable.*

Lemma 6.3 (Types are ordinary or splittable)**.** *For any type A, either A is ordinary or A is splittable, and it is decidable.*

Lemma 6.4 (Splittable determinism)**.** *For any splittable type A, if $B_1 \lhd A \rhd C_1$ and $B_2 \lhd A \rhd C_2$, then $B_1 = B_2$ and $C_1 = C_2$.*

Algorithmic BCD subtyping. By splitting a type into (nested) intersections of ordinary types, the distributivity rule in BCD subtyping is no longer problematic. In essence, we normalize the function type produced by distributivity to an equivalent intersection type.

Our new formulation of the subtyping relation $A <: B$ is shown at the bottom of Figure 11. The main idea with this formulation is that we always split B if possible. In such a case, rule S-bcd-and is applied, which works in a similar way to rule S-andR when D is already an intersection type, such as $D_1 \,\&\, D_2$. The most interesting case is when D is a splittable function type. For example, $D := D_1 \to (D_{21} \,\&\, D_{22})$, and D can be split into $D_1 \to D_{21}$ and $D_1 \to D_{22}$. Therefore, the premises of $A <: D$ are $A <: D_1 \to D_{21}$ and $A <: D_1 \to D_{22}$, or equivalently, $A <: (D_1 \to D_{21}) \,\&\, (D_1 \to D_{22})$, which can conclude $A <: D$ with a combination of rule OS-TRANS and rule OS-DISTARR in the declarative BCD subtyping. In fact, the split of two types already takes rule OS-DISTARR into consideration implicitly, while rule S-BCD-AND combines rule OS-TRANS and rule OS-AND. All the other rules are straightforward, because we already rule out the possibility that B is splittable. They look almost identical to standard subtyping rules found in the literature, modulo the additional ordinary-type conditions marked in gray.

Top-like types. Top-like types are the same as defined in Figure 3. Rule S-BCD-TOP says that a top-like type is a supertype of any type, which is equivalent to the declarative rule OS-TOP and rule OS-TOPARR. Although the supertype in rule OS-TOPARR looks different than that of rule TL-ARR, the equivalence is supported by the transitivity rule. For example, $\mathsf{Int} \to \mathsf{Top}$ and $\mathsf{Int} \to (\mathsf{Top} \to \mathsf{Top})$ are supertypes (and also subtypes) of Top. The following property generalizes rule OS-TOP in the declarative BCD subtyping, and it shows the soundness of the definition of top-like types.

Lemma 6.5 (S-BCD-TOP in declarative BCD). *If* $\rceil B \lceil$*, then* $A \leqslant B$.

Modularity. A more declarative (and modular) formulation of subtyping is to omit each ordinary-type condition in a gray background in Figure 11. Note that here we employ the term “modularity” to mean that existing subtyping rules do not need to be changed because of a new feature (in this case distributivity).

Our first observation is that omitting the ordinary-type conditions does not change expressiveness.

Lemma 6.6 (Modular S-BCD-TOP). *If* $\rceil B \lceil$*, then* $A <: B$.

Lemma 6.7 (Modular S-BCD-ARR). *If* $B <: A$*,* $C <: D$*, then* $A \to C <: B \to D$.

Lemma 6.8 (Modular S-BCD-ANDL). *If* $A <: C$*, then* $A \,\&\, B <: C$.

Lemma 6.9 (Modular S-BCD-ANDR). *If* $B <: C$*, then* $A \,\&\, B <: C$.

With these lemmas, the two formulations (with and without ordinary-type conditions) are proved to be sound and complete with respect to each other. Thus, compared to the subtyping relation in Figure 3 (which is not BCD), the modular BCD subtyping relation only replaces rule S-ANDR by rule S-BCD-AND to enable BCD distributivity. The new subtyping rules generalize the previous ones.

It is possible to have an equivalent alternative approach for adding BCD distributivity (rule S-BCD-AND) without modifying the existing rules. One just needs to keep the old rule S-ANDR and add rule S-BCD-AND-ALT:

$$\textsc{S-AndR}\quad \frac{A_1 <: A_2 \qquad A_1 <: A_3}{A_1 <: A_2 \,\&\, A_3} \qquad\qquad \textsc{S-BCD-And-Alt}\quad \frac{B \lhd E \rhd C \qquad A <: D \to B \qquad A <: D \to C}{A <: D \to E}$$

Additionally, top-like types are handled by rule S-BCD-TOP using the top-like relation ($\rceil A \lceil$). An alternative to that rule is to use the following two rules:

$$\frac{}{A <: \mathrm{Top}}\ \textsc{S-bcd-top-alt} \qquad \frac{\mathrm{Top} <: C}{A <: B \to C}\ \textsc{S-bcd-top-alt-arr}$$

The first rule is just the standard rule for top types, while the second rule is a special rule which deals with top-like function types.

Both alternative approaches replace one rule in our modular subtyping relation by two, while keeping the expressiveness of subtyping unchanged. Rule S-BCD-AND and rule S-BCD-TOP in our modular BCD subtyping are generalizations of the designs that would use 2 rules instead, which is why we choose them to be in our system.

It is also worth mentioning that our algorithmic relation keeps the simple judgment form $A <: B$, thus the system is easier to extend with orthogonal features, which have been presented with a subtyping relation of that form. Some BCD subtyping formulations require a different form to the subtyping relation (Bi *et al.*, 2018; Pierce, 1989; Bessai *et al.*, 2016, 2019).

6.3 Metatheory of modular BCD

A benefit of our new formulation of BCD subtyping is that the metatheory is remarkably simple. The metatheory of BCD subtyping has been a notoriously difficult topic of research.

Inversion lemmas. Given that our algorithmic relations are not entirely syntax-directed, several inversion lemmas indicate that the algorithm and the declarative system behave similarly.

Lemma 6.10 (Inversion on left split)**.** *Given B Ordinary$^+$, if $A <: B$ and $A_1 \lhd A \rhd A_2$ then $A_1 <: B$ or $A_2 <: B$.*

Lemma 6.11 (Inversion on right split)**.** *If $A <: B$ and $B_1 \lhd B \rhd B_2$ then $A <: B_1$ and $A <: B_2$.*

Both lemmas are easily proven by induction on the subtyping premises.

Transitivity. Since the transitivity rule is eliminated in algorithmic systems, we need to show that the transitivity lemma holds. This property is critical but difficult for any BCD formulation without the transitivity axiom built-in.

Lemma 6.12 (Transitivity of modular BCD)**.** *If $A <: B$ and $B <: C$ then $A <: C$.*

To prove the transitivity lemma, one might try at first to proceed by induction on B. However, that does not succeed, since our algorithm is not entirely syntax-directed. In particular, the behavior of the subtyping algorithm is determined by whether the type on the right is ordinary or splittable (but not simply the syntax form of the type). For example, in the case where $A <: B$ is derived by rule S-BCD-AND, B can be split into two

$\boxed{\vdash_\& A}$ *(Proper Types)*

$$\frac{}{\vdash_\& \mathsf{Int}}\ \textsc{rty-int} \qquad \frac{}{\vdash_\& \mathsf{Top}}\ \textsc{rty-top} \qquad \frac{B\ \mathsf{Ordinary}^{+} \quad \vdash_\& A \quad \vdash_\& B}{\vdash_\& A \to B}\ \textsc{rty-ordFun}$$

$$\frac{B \lhd A \rhd C \quad \vdash_\& B \quad \vdash_\& C}{\vdash_\& A}\ \textsc{rty-split}$$

Fig. 12. Proper types.

parts $B_1 \lhd B \rhd B_2$, yet B_1 and B_2 cannot be applied to the induction hypothesis, simply because they may not be components of type B. On the other hand, assuming one does induction on the two subtyping derivations, it is a tricky case when $A <: B$ is derived by rule S-BCD-AND and $B <: C$ is derived by rule S-BCD-ARR. The former splits type B while the latter decomposes it as a function type, and they do not match.

To overcome this problem we would like to treat any splittable type similarly to an intersection type. Therefore, we need a proper characterization of the type structure, so that the induction hypothesis on splittable types is always as desired. The relation defined in Figure 12 defines the so-called *proper types*. Proper types act as an alternative inductive definition for types, distinguishing types based on whether they are ordinary or splittable. The following lemma shows that the definition is general: any type is a proper type.

Lemma 6.13 (Types are proper types). *For any type A, $\vdash_\& A$.*

With the new definition for types, we are ready to prove the transitivity lemma. Induction is performed on the relation $\vdash_\& B$ which is obtained easily on type B through Lemma 6.13. The induction then breaks into several cases:

- Int and Top are easy base cases.
- When B is a function type constructed by rule RTY-ORDFUN, a nested induction on the premise $B <: C$ gives three sub-cases.
  - Sub-cases rule S-BCD-TOP and rule S-BCD-AND are easy to prove by induction hypothesis.
  - Sub-case rule S-BCD-ARR is then able to finish by another nested induction on the other premise $A <: B$.
- The last case is when B is a splittable type ($B_1 \lhd B \rhd B_2$), where we know that $A <: B_1$ and $A <: B_2$ by Lemma 6.11. Let us do induction on $\vdash_\& C$.
  - If C is an ordinary type, by Lemma 6.10, either $B_1 <: C$ or $B_2 <: C$ holds. In both cases, applying the induction hypothesis of B finishes the proof.
  - Otherwise, assuming $C_1 \lhd C \rhd C_2$, we apply Lemma 6.11 to $B <: C$ and get $B <: C_1$ and $B <: C_2$. Via the induction hypothesis of C we can obtain $A <: C_1$ and $A <: C_2$ and reach the goal by rule S-BCD-AND.

Equivalence to declarative BCD. Thanks to the simple judgment form used in our algorithm, the soundness and completeness theorems are stated directly as follows.

Theorem 6.1 (Soundness of modular BCD)**.** *If $A <: B$ then $A \leqslant B$.*

The soundness theorem only relies on Lemma 6.1 and Lemma 6.5.

The completeness theorem is also easy to show with the help of transitivity (Lemma 6.12), by induction on the premise.

Theorem 6.2 (Completeness of modular BCD)**.** *If $A \leqslant B$ then $A <: B$.*

To sum up, our novel formulation of BCD subtyping adds the function distributivity feature in a modular way, and the metatheory is straightforward to establish with the notion of proper types.

7 The nested composition calculus: Syntax, subtyping and typing

In this section, we will introduce the λ_i^+ calculus, which is essentially a TDOS variant of the calculus with the same name introduced by Bi *et al.* (2018). In the original λ_i^+ calculus, the semantics is defined by elaboration. λ_i^+ has record types and supports record concatenation via the merge operator. While λ_i^+ is quite similar to the λ_i calculus, BCD subtyping empowers λ_i^+ so that a merge of functions (or records) can act as a function (or a record). We will see how this behavior leads to changes in the typing and reduction rules.

7.1 Syntax and typing

The syntax of λ_i^+ is:

Types	A, B	$::=$	$\mathsf{Int} \mid \mathsf{Top} \mid A \to B \mid A \,\&\, B \mid \{l : A\}$
Expressions	e	$::=$	$x \mid i \mid \top \mid e : A \mid e_1\ e_2 \mid \lambda x.e : A \to B \mid e_1 \,,,\, e_2 \mid \mathsf{fix}\, x.\, e : A$ $\mid \{l = e\} \mid e.l$
Values	v	$::=$	$i \mid \top \mid \lambda x.e : A \to B \mid v_1 \,,,\, v_2 \mid \{l = v\}$
Pre-values	u	$::=$	$v \mid e : A \mid u_1 \,,,\, u_2 \mid \{l = u\}$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Values or labels	vl	$::=$	$v \mid \{l\}$

Records and record types. The addition of records affects definitions at both the term and type level. $\{l = e\}$ stands for a single-field record whose label is l and its field is e. Projection ($e.l$) selects the field(s) from e with label l. In a record type $\{l : A\}$, A is the type of the field. As discussed in Section 2, records can be concatenated by the merge operator. A merge of single-field records can be thought of as a multi-field record, and therefore can be used, for example, to model objects. Finally, a new syntactic category vl is defined to unify values and labels for reduction.

Splittable types and subtyping. Figure 13 shows the extension of ordinary types, splittable types, top-like types, and the modular BCD subtyping to record types. The original definitions can be found in Figure 11, discussed in Section 6. For each relation, a rule for record types is added in a modular way. Distributivity of intersection over record types is supported via rule S-BCD-AND. The definition of splittable types is extended with a new rule SP-RCD, which states that a record type is splittable if the type of its field is splittable.

$\boxed{A \text{ Ordinary}}$ *(Ordinary Types)*

$$\textsc{O-bcd-rcd}\quad \frac{B \text{ Ordinary}^{+}}{\{l : B\} \text{ Ordinary}^{+}}$$

$\boxed{\rceil A \lceil}$ *(Top-like Types)*

$$\textsc{TL-rcd}\quad \frac{\rceil B \lceil}{\rceil \{l : B\} \lceil}$$

$\boxed{B \lhd A \rhd C}$ *(Splittable Types)*

$$\textsc{Sp-rcd}\quad \frac{C \lhd B \rhd D}{\{l : C\} \lhd \{l : B\} \rhd \{l : D\}}$$

$\boxed{A <: B}$ *(Modular BCD Subtyping)*

$$\textsc{S-bcd-rcd}\quad \frac{\begin{array}{c} D \text{ Ordinary}^{+} \\ C <: D \end{array}}{\{l : C\} <: \{l : D\}}$$

Fig. 13. Rules extending subtyping in Figure 11 with records.

Disjointness. The disjointness definition, presented on the top of Figure 14, extends λ_i's definition in Figure 3. It might be a bit surprising that, except for the new record related rules, the remaining rules are the same as λ_i's disjointness definition. The two systems both respect the specification of disjointness (Definition 3.1), from which we know that if type A is not disjoint with type B, then it is not disjoint to any subtypes of B. Therefore, since types in λ_i^+ can have more supertypes, its disjointness definition is expected to be stricter than λ_i. However, in λ_i, for arrow types, disjointness only cares about output types. In other words, the set of all output types in an intersection of arrow types decides the set of its disjoint types. For $(A \to B) \,\&\, (A \to C)$, if a type is disjoint to it, the type cannot contain B or C in the return type of any of its components. The same criterion applies to types of disjoints from $(A \to B \,\&\, C)$. Therefore, for $(A \to B) \,\&\, (A \to C)$, the additional supertype $A \to B \,\&\, C$, introduced by the distributivity rule in BCD subtyping, brings no extra *non-disjoint* type to it. Thus, the disjointness definition does not change. What is more, the extended definition also has the following properties:

Lemma 7.1 (Disjointness properties)**.** *Disjointness satisfies:*

1. *$A * B$ if and only if $A *_a B$.*
2. *if $A * (B_1 \to C)$ then $A * (B_2 \to C)$.*
3. *if $A * B \,\&\, C$ then $A * B$ and $A * C$.*

Pre-values and consistency. As shown in Section 3.5, merges like $e : A \,,,\, e : A$, could be produced during reduction, since merged functions both take the input. To type check such merges, in λ_i^+, we use *pre-values* to denote a sort of terms including values, annotated terms, and merges composed by them, and generalize consistency to pre-values. A pre-value's type, if it is not a merge, can be told directly from its form without analyzing its structure. The *principal type* of a term is the most specific one among all of its types, i.e. it is the subtype of every other type of the term. The middle of Figure 14 shows the syntax-directed definition of principal types for pre-values. It is proved that for a well-typed pre-value with type A, its principal type is A.

Lemma 7.2 (Principal types)**.** *For any pre-value u,*

1. *if $u : A$ and $\cdot \vdash u \Rightarrow B$, then $A = B$.*
2. *if $\cdot \vdash u \Rightarrow A$ then $u : A$.*

$\boxed{A *_a B}$ *(Algorithmic Disjointness (Extended with Records))*

$$\textsc{D-RcdEq}\ \frac{A *_a B}{\{l : A\} *_a \{l : B\}} \qquad \textsc{D-RcdNeq}\ \frac{l_1 \neq l_2}{\{l_1 : A\} *_a \{l_2 : B\}} \qquad \textsc{D-IntRcd}\ \frac{}{\mathsf{Int} *_a \{l : A\}} \qquad \textsc{D-RcdInt}\ \frac{}{\{l : A\} *_a \mathsf{Int}}$$

$$\textsc{D-ArrRcd}\ \frac{}{A_1 \to A_2 *_a \{l : A\}} \qquad \textsc{D-RcdArr}\ \frac{}{\{l : A\} *_a A_1 \to A_2}$$

$\boxed{u : A}$ *(Principal Type of Pre-Values)*

$$\textsc{PT-Top}\ \frac{}{\top : \mathsf{Top}} \qquad \textsc{PT-Int}\ \frac{}{i : \mathsf{Int}} \qquad \textsc{PT-Lam}\ \frac{}{(\lambda x.\, e : A \to B) : (A \to B)} \qquad \textsc{PT-Rcd}\ \frac{u : A}{\{l = u\} : \{l : A\}}$$

$$\textsc{PT-Merge}\ \frac{u_1 : A \qquad u_2 : B}{(u_1 ,, u_2) : (A \,\&\, B)} \qquad \textsc{PT-Anno}\ \frac{}{(e : A) : A}$$

$\boxed{u_1 \approx u_2}$ *(Consistency)*

$$\textsc{C-Lit}\ \frac{}{i \approx i} \qquad \textsc{C-Abs}\ \frac{}{\lambda x.\, e : A \to B_1 \approx \lambda x.\, e : A \to B_2} \qquad \textsc{C-Anno}\ \frac{}{e : A \approx e : B} \qquad \textsc{C-Rcd}\ \frac{u_1 \approx u_2}{\{l = u_1\} \approx \{l = u_2\}}$$

$$\textsc{C-Disjoint}\ \frac{u_1 : A \qquad u_2 : B \qquad A *_a B}{u_1 \approx u_2} \qquad \textsc{C-MergeL}\ \frac{u_1 \approx u \qquad u_2 \approx u}{u_1 ,, u_2 \approx u} \qquad \textsc{C-MergeR}\ \frac{u \approx u_1 \qquad u \approx u_2}{u \approx u_1 ,, u_2}$$

Fig. 14. Extension of disjointness in Figure 3, principal types and consistency in λ_i^+.

Recall that the intuition of consistency is to allow two terms in a merge if they have disjoint types or their overlapped parts are equal. In λ_i, only values can be consistent, and the specification of consistency relies on typed reduction, which is hard to extend to expressions. To extend consistency to pre-values, we now use an inductive relation to define consistency, where principal types are used to simplify the definition. Consistency is showed on the bottom of Figure 14. Notably, for values, the definition is sound and complete with respect to the specification (Definition 3.2).

Lemma 7.3 (Soundness and completeness of consistency definition)**.** *For all well-typed value v_1 and v_2, $v_1 \approx v_2$ if and only if $v_1 \approx_{spec} v_2$.*

Typing and applicative distributivity. Figure 15 presents the extension of typing and applicative distributivity. The initial definitions can be found in Figure 4. Applicative distributivity is extended in two dimensions. One is to treat record types as one of the applicative forms. Rule AD-Rcd is the base case for record types, given that the shape can already accept label projections. Top can behave like different types depending on the context: 1) $\{l : \mathsf{Top}\}$ when a record type is required, or 2) $\mathsf{Top} \to \mathsf{Top}$ when a function type is required. The other dimension is about distributivity over intersections, where rule AD-AndArr now supports intersection of function (-like) types, thanks to the distributivity rule of BCD subtyping. Rule AD-AndRcd combines both dimensions and enables the intersection of record (-like) types.

$\boxed{A \rhd B}$ *(Applicative Distributivity (λ_i^+ Extension))*

$$\textsc{AD-rcd}\quad \frac{}{\{l : A\} \rhd \{l : A\}} \qquad \textsc{AD-topRcd}\quad \frac{}{\mathsf{Top} \rhd \{l : \mathsf{Top}\}} \qquad \textsc{AD-andRcd}\quad \frac{A \rhd \{l : A_2\} \qquad B \rhd \{l : B_2\}}{A \,\&\, B \rhd \{l : A_2 \,\&\, B_2\}}$$

$$\textsc{AD-andArr}\quad \frac{A \rhd A_1 \to A_2 \qquad B \rhd B_1 \to B_2}{A \,\&\, B \rhd A_1 \,\&\, B_1 \to A_2 \,\&\, B_2}$$

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ *(Bidirectional Typing (λ_i^+ Extension))*

$$\textsc{Typ-bcd-rcd}\quad \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\}} \qquad \textsc{Typ-bcd-proj}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad A \rhd \{l : C\}}{\Gamma \vdash e.l \Rightarrow C} \qquad \textsc{Typ-bcd-mergev}\quad \frac{\begin{array}{c}\cdot \vdash u_1 \Rightarrow A \\ \cdot \vdash u_2 \Rightarrow B \qquad u_1 \approx u_2\end{array}}{\Gamma \vdash u_1 \,,,\, u_2 \Rightarrow A \,\&\, B}$$

Fig. 15. Typing and applicative distributivity of λ_i^+ (extends Figure 4).

Typing relies on applicative distributivity. Due to the distributivity and top-like types, in rule Typ-app and rule Typ-bcd-proj, where a term is expected to play the role of a function or record, its inferred type is allowed to be Top, or (in λ_i^+) an intersection type. Assuming that a term e_1 of type $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Bool} \to \mathsf{Bool})$ is applied to term e_2, via this relation, we can derive that e_2 should be checked against type $\mathsf{Int} \,\&\, \mathsf{Bool}$. Besides this, two new rules are added for records and record projection: rule Typ-bcd-rcd and rule Typ-bcd-proj. Moreover, rule Typ-bcd-mergev is generalized from values to *pre-values*. Thus, merges like $e : A \,,,\, e : A$ are well typed in λ_i^+.

7.2 Operational semantics

Typed reduction. Compared with λ_i, the new typed reduction has one more rule for records, and two rules that change, as shown in Figure 16. An additional condition is added in rule TR-bcd-arrow to make sure that it only applies to ordinary arrow types. The condition is unnecessary in λ_i because every arrow type there is ordinary. Rule TR-bcd-rcd mimics the arrow rule. Rule TR-bcd-and works on splittable types, so now it needs to take care of more types than just intersections. Although we choose to merge the two results of the split types, there are some other alternative options. One possible design is to construct a lambda from the results. Therefore, we could prevent merges from being the inhabitants of arrow types. However, manipulating the lambda body breaks the transitivity of typed reduction, which plays an important role in our metatheory.

Reduction. In the reduction rules of λ_i^+, presented in Figure 17, rule Step-bcd-papp replaces the original beta-reduction rule and rule Step-top. Rule Step-bcd-rcd, rule Step-bcd-proj, and rule Step-bcd-pproj are added for records and record projection. The rules for merges are changed to reduce components in parallel.

Parallel application. The distributivity rule in BCD subtyping indicates that a merge of functions can be applied. While the current typing rule can check such applications with suitable annotations, designing new reduction rules is necessary. An intuitive solution is to

$\boxed{v \hookrightarrow_A v'}$ *(Typed Reduction (λ_i^+ Extension))*

$$\textsc{tr-bcd-rcd}\quad \frac{A\ \mathsf{Ordinary}^+ \qquad \neg\rceil A \lceil \qquad v \hookrightarrow_A v'}{\{l = v\} \hookrightarrow_{\{l:A\}} \{l = v'\}} \qquad\qquad \textsc{tr-bcd-arrow}\quad \frac{\begin{array}{c} D\ \mathsf{Ordinary}^+ \\ \neg\rceil D \lceil \qquad C <: A \qquad B <: D \end{array}}{\lambda x.\, e : A \to B \hookrightarrow_{(C \to D)} \lambda x.\, e : A \to D}$$

$$\textsc{tr-bcd-and}\quad \frac{B \lhd A \rhd C \qquad v \hookrightarrow_B v_1 \qquad v \hookrightarrow_C v_2}{v \hookrightarrow_A v_1 \,,,\, v_2}$$

Fig. 16. Typed reduction of λ_i^+ (extends Figure 6).

have a rule that distributes the input value, like

$$(v_1 \,,,\, v_2)\; v \hookrightarrow v_1\; v \,,,\, v_2\; v$$

Assuming that v_1 and v_2 are consistent but not disjoint, to obtain preservation, $v_1\ v$ and $v_2\ v$ have to be consistent. To avoid the complexity of extending consistency to expressions including applications, we design *parallel application* (Figure 17) to distribute and substitute the input value in a big-step style, where a function application is divided into two parts v and vl, and steps to an expression e. Consider a merge of three functions being applied to a value. Compared to adding the previous single rule to the small-step reduction, parallel reduction helps us to "jump" from $(f_1 \,,,\, f_2 \,,,\, f_3)\ v$ to a merge of annotated terms when reasoning about reduction. Every lambda gets the input directly without intermediate reduction steps such as $((f_1 \,,,\, f_2)\ v) \,,,\, f_3\ v$. Record projection is handled in a similar style. In this case, v is a record value, and vl stands for a label instead.

$$\{l = 1\} \,,,\, (\{l = \mathsf{True}\} \,,,\, \{l = 1\}) \bullet \{l\} \hookrightarrow 1 \,,,\, (\mathsf{True} \,,,\, 1)$$

The above example shows how merged records are projected in parallel, and the whole term is kept consistent. Rule PAPP-TOP shows that the top value can be used as a function that returns $\top$, or a record which contains $\top$ in its field. With it, rule STEP-BCD-PAPP subsumes the rule STEP-TOP in Figure 8.

Parallel reduction of merges. To maintain consistency of subterms in a merge (which may contain non-values), as required by the typing rule TYP-MERGEV, we reduce every component in a merge simultaneously through rule STEP-BCD-MERGE. This rule is helpful to preserve consistency of pre-values during reduction, and therefore enables the type preservation theorem. As a counter-example, if parallel reduction is not employed, we might encounter the following reduction step:

$$(1\ + 1) : \mathsf{Int} \,,,\, (1\ + 1) : \mathsf{Int} \hookrightarrow 2 : \mathsf{Int} \,,,\, (1\ + 1) : \mathsf{Int}$$

where the r.h.s term is ill-typed, since the subterms are not consistent any more. In contrast, rule STEP-BCD-MERGE will keep the reduction of terms synchronized. When one of the subterms is already a value, rule STEP-BCD-MERGE no longer applies. In that case, rule STEP-BCD-MERGEL or rule STEP-BCD-MERGER reduces the other subterm.

Overloading on return types. The λ_i and λ_i^+ calculi support a form of overloading on return types. With parallel application, we can even merge multiple functions and apply them together to one input. Eventually, the return value would be a combination of the

$\boxed{v \bullet vl \hookrightarrow e}$ *(Parallel Application)*

$$\textsc{Papp-abs}\ \frac{v \hookrightarrow_A v'}{\lambda x.\, e : A \to B \bullet v \hookrightarrow (e[x \mapsto v']) : B} \qquad \textsc{Papp-proj}\ \frac{}{\{l = v\} \bullet \{l\} \hookrightarrow v} \qquad \textsc{Papp-top}\ \frac{}{\top \bullet vl \hookrightarrow \top}$$

$$\textsc{Papp-merge}\ \frac{v_1 \bullet vl \hookrightarrow e_1 \qquad v_2 \bullet vl \hookrightarrow e_2}{(v_1 \,,,\, v_2) \bullet vl \hookrightarrow e_1 \,,,\, e_2}$$

$\boxed{e \hookrightarrow e'}$ *(Reduction (λ_i^+ Extension))*

$$\textsc{Step-bcd-papp}\ \frac{v_1 \bullet v_2 \hookrightarrow e}{v_1\, v_2 \hookrightarrow e} \qquad \textsc{Step-bcd-rcd}\ \frac{e \hookrightarrow e'}{\{l = e\} \hookrightarrow \{l = e'\}} \qquad \textsc{Step-bcd-proj}\ \frac{e \hookrightarrow e'}{e.l \hookrightarrow e'.l} \qquad \textsc{Step-bcd-pproj}\ \frac{v \bullet \{l\} \hookrightarrow v'}{v.l \hookrightarrow v'}$$

$$\textsc{Step-bcd-merge}\ \frac{e_1 \hookrightarrow e_1' \qquad e_2 \hookrightarrow e_2'}{e_1 \,,,\, e_2 \hookrightarrow e_1' \,,,\, e_2'} \qquad \textsc{Step-bcd-mergel}\ \frac{e_1 \hookrightarrow e_1'}{e_1 \,,,\, v_2 \hookrightarrow e_1' \,,,\, v_2} \qquad \textsc{Step-bcd-merger}\ \frac{e_2 \hookrightarrow e_2'}{v_1 \,,,\, e_2 \hookrightarrow v_1 \,,,\, e_2'}$$

Fig. 17. Parallel application and reduction of λ_i^+ (extends Figure 8).

outputs of all functions and can play the role of any single output. The following example shows how this mechanism works.

$$\begin{array}{ll}
 & \mathsf{not}\ ((\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int} \,,,\, \lambda x.\, \mathsf{True} : \mathsf{Int} \to \mathsf{Bool})\ 1) \\
\hookrightarrow & \{\text{ by } \textsc{Step-bcd-papp} \text{ and parallel application }\} \\
 & \mathsf{not}\ (2 : \mathsf{Int} \,,,\, \mathsf{True} : \mathsf{Bool}) \\
\hookrightarrow & \{\text{ by } \textsc{Step-appr}, \textsc{Step-bcd-merge}, \textsc{Step-annov} \text{ and typed reduction }\} \\
 & \mathsf{not}\ (2 \,,,\, \mathsf{True}) \\
\hookrightarrow & \{\text{ assuming } \mathsf{not} \text{ has type } \mathsf{Bool} \to \mathsf{Bool} \} \\
 & \mathsf{False} : \mathsf{Bool} \\
\hookrightarrow & \{\text{ by } \textsc{Step-annov} \text{ and typed reduction }\} \\
 & \mathsf{False}
\end{array}$$

7.3 Metatheory

Completeness of the type system with respect to the original λ_i^+ (or NeColus) calculus. Besides the extra rule for consistent merges (rule TYP-BCD-MERGEV), λ_i^+ has two different rules for record projection and function application when compared with the type system of NeColus (Bi *et al.*, 2018).

$\boxed{\Gamma \vdash_n e \Leftrightarrow A}$ *(NeColus Typing (Selected))*

$$\textsc{Nec-T-app}\ \frac{\Gamma \vdash_n e_1 \Rightarrow A_1 \to A_2 \qquad \Gamma \vdash_n e_2 \Leftarrow A_1}{\Gamma \vdash_n e_1\, e_2 \Rightarrow A_2} \qquad \textsc{Nec-T-proj}\ \frac{\Gamma \vdash_n e \Rightarrow \{l : A\}}{\Gamma \vdash_n e.l \Rightarrow A}$$

To show that every well-typed term in NeColus can be type checked in λ_i^+, we prove the following lemmas:

Lemma 7.4 (λ_i^+ **application subsumes NeColus's application).** *For any expressions e_1 and e_2, if $\Gamma \vdash e_1 \Rightarrow A \to B$ and $\Gamma \vdash e_2 \Leftarrow A$, then $\Gamma \vdash e_1\, e_2 \Rightarrow B$.*

$\boxed{A \ll B}$ *(BCD Runtime Subtyping (λ_i^+ Extension))*

$$\textsc{RSub-bcd-rcd}\quad \frac{A \ll B}{\{l : A\} \ll \{l : B\}} \qquad \textsc{RSub-bcd-top}\quad \frac{A\ \mathsf{Ordinary}^+ \qquad \rceil A \lceil}{\mathsf{Top} \ll A}$$

$$\textsc{RSub-bcd-split}\quad \frac{\begin{array}{c} A_1 \lhd A \rhd A_2 \\ B_1 \lhd B \rhd B_2 \qquad A_1 \ll B_1 \qquad A_2 \ll B_2 \end{array}}{A \ll B}$$

Fig. 18. Runtime subtyping of λ_i^+ extends the definition in Figure 7.

Lemma 7.5 (λ_i^+ projection subsumes NeColus's projection). *For any expressions e and any label l, if* $\Gamma \vdash e \Rightarrow \{l : A\}$, *then* $\Gamma \vdash e.l \Rightarrow C$.

Then it is straightforward that NeColus can be translated into λ_i^+. In our Coq formalization, we designed an elaboration from NeColus to λ_i^+ and proved the completeness of λ_i^+'s type system with respect to NeColus.

Properties of the TDOS. The TDOS of λ_i^+ preserves determinism, progress, and subject-reduction. Most of the proof follows λ_i's structure. Runtime subtyping is extended, and the newly added parallel application requires some extra lemmas.

Runtime subtyping and preservation. Compared with Figure 7, λ_i^+'s runtime subtyping has one more rule for record types, and two rules changed for distributivity (Figure 18). Rule RSub-bcd-rcd allows a record type to be a runtime subtype of another record type if their label is the same and the former's field type is a runtime subtype of the latter's. Type A is constrained to be ordinary in rule RSub-bcd-top. Therefore, $\mathsf{Top} \ll \mathsf{Top}\ \&\ \mathsf{Top}$ and $A \to \mathsf{Top} \ll A \to \mathsf{Top}\ \&\ \mathsf{Top}$ are no longer derivable. The change is to maintain the transitivity of subtyping. As a consequence of the generalization of rule RSub-and to rule RSub-bcd-split, an intersection type can be a runtime supertype of an arrow type. However, $A \to \mathsf{Top}$, as an ordinary type, is not a runtime subtype of $(A \to \mathsf{Top})\ \&\ (A \to \mathsf{Top})$, and that would break transitivity. Since we need rule RSub-bcd-split to help some merges of functions to act as a function, we choose to drop the unneeded $A \to \mathsf{Top} \ll A \to \mathsf{Top}\ \&\ \mathsf{Top}$.

Parallel application. Some extra lemmas about parallel application are proved:

Lemma 7.6 (Type preservation of parallel application on functions). *If* $\cdot \vdash v_1\ v_2 \Rightarrow A$, *and* $v_1 \bullet v_2 \hookrightarrow e$ *then* $\cdot \vdash e \Rightarrow A$.

Lemma 7.7 (Type preservation of parallel application on records). *If* $\cdot \vdash v_1.l \Rightarrow A$, *and* $v_1 \bullet \{l\} \hookrightarrow e$ *then* $\cdot \vdash e \Rightarrow A$.

For both function application and record projection, parallel reduction preserves the original type. Furthermore, we can prove the following determinism lemmas:

Lemma 7.8 (Determinism of parallel application on functions). *If* $\cdot \vdash v_1\ v_2 \Rightarrow A$, $v_1 \bullet v_2 \hookrightarrow e_1$, $v_1 \bullet v_2 \hookrightarrow e_2$ *then* $e_1 = e_2$.

Lemma 7.9 (Determinism of parallel application on records). *If* $\cdot \vdash v_1.l \Rightarrow A$*,* $v_1 \bullet l \hookrightarrow e_1$*,* $v_1 \bullet l \hookrightarrow e_2$ *then* $e_1 = e_2$*.*

We can also prove the following progress lemmas for parallel application:

Lemma 7.10 (Progress of parallel application on functions). *If* $\cdot \vdash v_1\ v_2 \Rightarrow A$*, then* $\exists e$*,* $v_1 \bullet v_2 \hookrightarrow e$*.*

Lemma 7.11 (Progress of parallel application on records). *If* $\cdot \vdash v_1.l \Rightarrow A$*, then* $\exists e$*,* $v_1 \bullet l \hookrightarrow e$*.*

Finally, based on all the lemmas above, the key properties of reduction can be derived, including type preservation with runtime subtyping:

Lemma 7.12 (Type preservation of $\hookrightarrow$ with respect to runtime subtyping). *If* $\cdot \vdash e \Rightarrow A$*, and* $e \hookrightarrow e'$ *then exists* B*,* $\cdot \vdash e' \Rightarrow B$ *and* $B \ll A$*.*

And its corollary:

Theorem 7.1 (Type preservation of $\hookrightarrow$). *If* $\cdot \vdash e \Rightarrow A$*, and* $e \hookrightarrow e'$ *then* $\cdot \vdash e' \Leftarrow A$*.*

Determinism and progress theorems are proved as well.

Theorem 7.2 (Determinism of $\hookrightarrow$). *If* $\cdot \vdash e \Rightarrow A$*,* $e \hookrightarrow e_1$*,* $e \hookrightarrow e_2$*, then* $e_1 = e_2$*.*

Theorem 7.3 (Progress of $\hookrightarrow$). *If* $\cdot \vdash e \Rightarrow A$*, then* e *is a value or* $\exists e'$*,* $e \hookrightarrow e'$*.*

8 Discussion

This section provides some discussion on design choices, a comparison with elaboration semantics as well as implementation considerations and possible extensions for future work.

8.1 TDOS versus an elaboration semantics

This paper proposes the use of type-directed operational semantics for modelling languages with a merge operator. Since a general form of the merge operator has a type-directed semantics, previous work has favored an elaboration semantics. Traditionally, for languages with type-directed semantics, elaboration has been a common choice. That is the case, for instance, for many previous calculi with the merge operator (Dunfield, 2014; Oliveira *et al*., 2016; Alpuim *et al*., 2017; Bi *et al*., 2018, 2019), *gradual typing* (Siek & Taha, 2006), or *type classes* (Wadler & Blott, 1989; Kaes, 1988). In the elaboration approach, the idea is that the source language can be translated via a type-directed translation into a conventional target calculus, whose semantics is not type directed. In the case of languages with the merge operator, the target calculus is typically a conventional calculus (such as the STLC) extended with pairs. The implicit upcasts that extract components from merges are modelled by explicit projections with pairs. One very appealing benefit of the elaboration semantics is that it gives a simple way to obtain an implementation. Since elaboration targets conventional languages/calculi, that means that it can simply reuse existing and efficient implementations of languages. For example, in the case of the merge operator, a key motivation of Dunfield (2014) was that the elaboration could just target ML-like languages, which have several efficient implementations available. From the implementation point-of-view, the TDOS would be more useful for guiding the design of a dedicated virtual machine, compiling directly to bytecode/assembly, or having an interpreter. However, obtaining an efficient implementation for the latter options would require significantly more effort than with an elaboration approach.

Nevertheless, the reason for designing a semantics for a language or calculus is not merely implementation. In fact, the implementation aspects are not our primary motivation for TDOS. Although, as we will discuss in Section 8.2, TDOS can provide some insights for obtaining efficient implementations as well. Furthermore, we do not view TDOS as being mutually exclusive with elaboration: both approaches have interesting aspects and they can serve different purposes. Our main motivation for the TDOS is reasoning. A semantics is supposed to describe the meaning of the language constructs in the language. Having a clear and high-level presentation of the semantics is then useful for language implementers to understand the behavior of the language. Furthermore, it is also useful to provide programmers with a mental model of how programs are executed, and to study the properties of the language. Next, we give more details on the advantages of a direct semantics over the elaboration semantics in terms of reasoning and proof methods employed in previous work on disjoint intersection types.

Shorter, more direct reasoning. Programmers want to understand the meaning of their programs. A formal semantics can help with this. With our TDOS, we can essentially employ a style similar to equational reasoning in functional programming to directly reason about programs written in λ_i. For example, it takes a few reasoning steps to work out the result of $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\,(2\,,,\,\text{‘}c\text{’})$:

$$\begin{array}{lll} & (\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\,(2\,,,\,\text{‘}c\text{’}) & \\ \hookrightarrow & (2 + 1) : \mathsf{Int} & \text{by } \textsc{Step-beta} \text{ and typed reduction} \\ \hookrightarrow & 3 : \mathsf{Int} & \text{by } \textsc{Step-anno} \text{ and arithmetic} \\ \hookrightarrow & 3 & \text{by } \textsc{Step-annov} \text{ and typed reduction} \end{array}$$

Here reasoning is easily justifiable from the small-step reduction rules and type-directed reduction. Building tools (such as debuggers) that automate such kind of reasoning should be easy using the TDOS rules.

However, with an elaboration semantics, the (precise) reasoning steps to determine the final result are more complex. First, the expression has to be translated into the target language before reducing to a similar target term. Figure 19 shows this elaboration process in λ_i, where an expression in the source language is translated into an expression in a target language with products. The source term $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\,(2\,,,\,\text{‘}c\text{’})$ is elaborated into the target term $(\lambda x.\, x + 1)\,(\mathsf{fst}\,(2\,,\,\text{‘}c\text{’}))$. As we can see the actual derivation is rather long, so we skip the full steps. Also, for simplicity's sake, here we assume the subtyping judgement produces the most straightforward coercion fst. This elaboration step and the introduction of coercions into the program make it harder for programmers to precisely understand the semantics of a program. Moreover, while the coercions inserted in this small expression may not look too bad, in larger programs the addition of coercions can be a lot more severe, hampering the understanding of the program.

$$\textsc{T-App}\ \dfrac{\textsc{T-Ann}\ \dfrac{\cdots}{\cdot \vdash (\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int}) \Rightarrow \mathsf{Int} \to \mathsf{Int} \rightsquigarrow \lambda x.\, x + 1} \quad \textsc{T-Sub}\ \dfrac{\textsc{T-Merge}\ \dfrac{\cdots}{\cdot \vdash (2 \,,,\, \text{‘}c\text{’}) \Rightarrow \mathsf{Int} \,\&\, \mathsf{Char} \rightsquigarrow (2 \,,\, \text{‘}c\text{’})} \quad \begin{array}{l}\textsc{Sub-AndL}\\ \mathsf{Int} \,\&\, \mathsf{Char} <: \mathsf{Int} \rightsquigarrow \mathsf{fst}\end{array}}{\cdot \vdash (2 \,,,\, \text{‘}c\text{’}) \Leftarrow \mathsf{Int} \rightsquigarrow \mathsf{fst}\ (2 \,,\, \text{‘}c\text{’})}}{\cdot \vdash (\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\ (2 \,,,\, \text{‘}c\text{’}) \Rightarrow \mathsf{Int} \rightsquigarrow (\lambda x.\, x + 1)\ (\mathsf{fst}\ (2 \,,\, \text{‘}c\text{’}))}$$

Fig. 19. Elaboration of $(\lambda x.\, x + 1 : \mathsf{Int} \to \mathsf{Int})\ (2 \,,,\, \text{‘}c\text{’})$ to a calculus with products.

After elaboration, we can then use the target language semantics, to determine a target language value.

$$\begin{array}{lll} & (\lambda x.\, x + 1)\ (\mathsf{fst}\ (2 \,,\, \text{‘}c\text{’})) & \\ \hookrightarrow & (\lambda x.\, x + 1)\ 2 & \text{reduction for application and pairs} \\ \hookrightarrow & 2 + 1 & \text{by the beta reduction rule} \\ \hookrightarrow & 3 & \text{by arithmetic} \end{array}$$

A final issue is that sometimes it is not even possible to translate back the value of the target language into an equivalent "value" on the source. For instance in the NeColus calculus (Bi *et al.*, 2018) $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ results in $(1, 1)$, which is a pair in the target language. But the corresponding source value $1 \,,,\, 1$ is not typable in NeColus. In essence, with an elaboration, programmers must understand not only the source language but also the elaboration process as well as the semantics of the target language, if they want to precisely understand the semantics of a program. Since the main point of semantics is to give clear and simple rules to understand the meaning of programs, a direct semantics is a better option for providing such understanding.

Simpler proofs of unambiguity. For calculi with an elaboration semantics, unrestricted intersections make it harder to prove coherence. Our λ_i calculus, on the other hand, has a deterministic semantics, which implies unambiguity directly. For instance, $(1 : \mathsf{Int} \,\&\, \mathsf{Int}) : \mathsf{Int}$ only steps to 1 in λ_i. But it can be elaborated into two target expressions in the NeColus calculus corresponding to two typing derivations:

$$(1 : \mathsf{Int} \,\&\, \mathsf{Int}) : \mathsf{Int} \rightsquigarrow \mathsf{fst}\ \ (1, 1)$$
$$(1 : \mathsf{Int} \,\&\, \mathsf{Int}) : \mathsf{Int} \rightsquigarrow \mathsf{snd}\ \ (1, 1)$$

Thus,the coherence proof needs deeper knowledge about the semantics: the two different terms are known to both reduce to 1 eventually. Therefore, they are related by the logical relation employed in NeColus for coherence. Things get more complicated for functions. The following example shows two possible elaborations of the same function. Relating them requires reasoning inside the binders and a notion of contextual equivalence.

$$\lambda x.\, x + 1 : \mathsf{Int} \,\&\, \mathsf{Int} \to \mathsf{Int} \rightsquigarrow \lambda x.\, \mathsf{fst}\ \ x + 1$$
$$\lambda x.\, x + 1 : \mathsf{Int} \,\&\, \mathsf{Int} \to \mathsf{Int} \rightsquigarrow \lambda x.\, \mathsf{snd}\ \ x + 1$$

Furthermore, the two target expressions above are *clearly not equivalent* in the general case. For instance, if we apply them to $(1, 2)$ we get different results. However, the target expressions will always behave equivalently when applied to arguments *elaborated from* *the* NeColus *source calculus*. NeColus, forbids terms like $(1 ,, 2)$ and thus cannot produce a target value $(1, 2)$. Because of elaboration and also this deeper form of reasoning required to show the equivalence of semantics, calculi defined by elaboration require a lot more infrastructure for the source and target calculi and the elaboration between them, while in a direct semantics only one calculus is involved and the reasoning required to prove determinism is quite simple.

Not limited to terminating programs. The (basic) forms of logical relations employed by NeColus and F_i^+ has cannot deal with non-terminating programs. In principle, recursion could be supported by using a *step-indexed logical relation* (Ahmed, 2006), but this is left for future work. λ_i smoothly handles unrestricted intersections and recursion, using TDOS to reach determinism with a significantly simpler proof method. It also makes other features that lead to nonterminating programs, such as *recursive types*, feasible.

8.2 Implementation considerations

The TDOS for λ_i and λ_i^+ is implementable directly, since the relations developed in this paper are all essentially algorithmic. However, a direct implementation will not be very efficient, for multiple reasons. Next we discuss some considerations for the design of efficient implementations of languages with the merge operator.

Merge lookups. From the efficiency point of view, one particularly bad aspect of the TDOS is that runtime lookup on merges (triggered by typed reduction) does not exploit statically known information. For instance, if we have a program such as

$$\mathsf{let}\ x : \ldots \,\&\, \mathsf{Int} \,\&\, \ldots\ =\ \ldots ,, 2 ,, \ldots\ \mathsf{in}\ x + 1$$

extracting the number 2 from the merge currently requires blindly going through the elements in the merge at runtime until the integer value is found. The approach in the elaboration semantics is much better here, since during type checking we know statically where the integer value can be found. Therefore, when generating code in the target language we can simply look up the value directly at that position, thus avoiding having to search for the value in the merge at runtime. It should be possible to create optimizations for the TDOS by employing similar ideas to the elaboration. That is, using the type system to statically determine where to find values. For this to work with the TDOS, we would need to extend the TDOS with explicit projections, so that code with implicit projections could be replaced with explicit projections. For instance, if the integer 2 is at position 5 in a merge, the code $x + 1$ could be replaced by $x[5] + 1$, where $x[5]$ denotes the projection of the 5th element in the merge. We should remark that, for such optimizations to be correct, it is essential that the origin of values can be statically determined. As we have discussed in Section 2.1, with some alternative semantics for the merge operator that may not be the case. Therefore, such optimizations would be invalid.

Overhead from annotations. One other source of concern for efficiency is the overhead caused by type annotations, and their use at runtime. Type annotations in a TDOS play a very similar role to casts in cast languages (Wadler & Findler, 2009; Siek & Wadler, 2010), which are used, for instance, as targets for gradually typed languages. The problem of how to design efficient cast languages is an important topic in the gradual typing literature, and the overhead caused by casts have been a notorious challenge in that area. In particular, a main source of overhead comes from casting functions. In calculi such as the blame calculus, function values need to accumulate casts to avoid raising blame too early. Thus, function values can have an arbitrary number of casts, which also leads to space efficiency concerns. Luckily, for λ_i and λ_i^+, we do not need to accumulate annotations around function values. Indeed, as shown in Section 4, function values are of the form $\lambda x.\, e : A \to B$ in λ_i and λ_i^+. That is, they only contain a single annotation. In contrast to cast calculi, the approach used in λ_i and λ_i^+, when reduction encounters multiple annotations around a function is to simply replace the type annotation of the value. This process happens in the typed reduction rule TR-ARROW for functions:

$$\textsc{TR-arrow}\quad \frac{\neg\rceil A_2 \lceil \qquad A_1 <: B_1 \qquad B_2 <: A_2}{\lambda x.\, e : B_1 \to B_2 \hookrightarrow_{(A_1 \to A_2)} \lambda x.\, e : B_1 \to A_2}$$

Moreover, as discussed in Section 5.2, the premise $A_1 <: B_1$ is not needed at runtime and can be avoided in an implementation. Thus, while the overhead caused by type annotations is still a concern, we believe that λ_i and λ_i^+ avoid some of the thornier issues that gradually typed languages have to deal with. Moreover, annotation replacement could actually have some advantages over an elaboration approach. Multiple annotations around functions would result in multiple coercions being applied to the function in an elaboration approach. In contrast, annotation replacement avoids such coercions and immediately collapses annotations, so that, ultimately, we only need to type-reduce the argument and the result of the function one time. Nevertheless, a proper assessment of the performance impact of annotations at runtime is outside of the scope of this work and left for future work.

Parallel application. Parallel application in λ_i^+ may raise some concern if available in unrestricted ways to programmers. In λ_i^+ parallel application arises as a consequence of the distributivity of intersections over functions. That is, the following is a valid subtyping statement in λ_i^+ (and BCD subtyping):

$$(A_1 \to B_1)\&(A_2 \to B_2) <: A_1\&A_2 \to B_1\&B_2$$

In essence, an intersection of two functions can be viewed as a single function with the intersections of the inputs and outputs. Thus, parallel application arises naturally in the semantics of the language in order to support such form of conversions at runtime.

The semantics for the application of merged functions is to apply all of the functions to the argument. This turns what looks like one function call into two or more function calls and could have a significant impact on the time complexity of a program. A program that looks linear time could in fact be exponential, due to parallel application. Therefore, it is important to consider the consequences of parallel application and how they can be mitigated. One option would be to redesign the calculus so that parallel application is avoided. This would probably require weakening the subtyping relation to avoid type conversions like the above. However, parallel application is an important aspect of nested composition, and dropping parallel application from the calculus would prevent important applications of nested composition. Instead, we believe that a better approach is to restrict the use of parallel application in the source language that targets λ_i^+. In the following section, we will come back to this topic.

8.3 Design of source languages

SEDEL is an existing implementation of a source language that targets calculi closely related to λ_i and λ_i^+. In essence the two implementations of SEDEL (Bi & Oliveira, 2018; Bi *et al.*, 2019) target F_i (Alpuim *et al.*, 2017) and F_i^+ (Bi *et al.*, 2019), which are, respectively, polymorphic versions of λ_i and λ_i^+ (but with an elaboration semantics). As already discussed in Section 2, the examples that we presented in this paper are written in a monomorphic subset of SEDEL and can be encoded into λ_i and λ_i^+ following the encoding proposed by Bi & Oliveira (2018).

While calculi like λ_i and λ_i^+ are supposed to be closer to source languages, they are still core calculi and are not meant to be used directly for programming. Here we discuss some important considerations for the design of source languages.

Type inference. λ_i and λ_i^+ employ explicit annotations at runtime to determine the final result of a program. However, while not too heavy, some of those annotations can be cumbersome to write explicitly by programmers. In λ_i^+, the rules of records essentially require explicit-type annotations when multiple distinct field names exist. For example, the following program is not directly accepted:

$$(\{m = 1\} ,, \{n = \mathsf{True}\}).n$$

Instead, to retrieve True, we need to write:

$$((\{m = 1\} ,, \{n = \mathsf{True}\}) : \{n : \mathsf{Bool}\}).n$$

The explicit annotation specifies both the field name and the type that we wish to look up. Note that, since records can have fields with the same names and disjoint types, having the type is helpful to select from those fields. For simplification, the SEDEL language accepts:

$$(\{m = 1\} ,, \{n = \mathsf{True}\}).n$$

directly, and inserts the missing annotation when elaborating to the core language. Once SEDEL finds a field lookup expression $e.n$, it infers the type of e and then filters that type using the label n. For the example above, the inferred type of record is

$$(\{m = 1\} ,, \{n = \mathsf{True}\}) : \{m : \mathsf{Int}\} \,\&\, \{n : \mathsf{Bool}\}$$

and the filtered type is $\{n : \mathsf{Bool}\}$. That annotation is inserted into the expression, resulting in:

$$((\{m = 1\} ,, \{n = \mathsf{True}\}) : \{n : \mathsf{Bool}\}).n$$

Parallel application again. Parallel application is a useful feature. It is used, for instance, in the example shown in Section 2.3, in the SEDEL code:

```
fac = new[ExtLang] implExt;
e = fac.add (fac.neg (fac.lit 2)) (fac.lit 3)
```

The add call above is employing parallel application and it will call the two implementations for add in the traits implPrint and implLang. Therefore, parallel applications play a fundamental role in "family polymorphism" style applications and cannot be entirely avoided without ruling out interesting applications. Nevertheless, as argued in Section 8.2, allowing a completely unrestricted form of parallel application may be undesirable. One possibility is to restrict the use of parallel applications in the source language. For instance, we may consider not allowing merges directly in the source language, and only use them to encode source-level features. In the interests of first-class traits with nested composition, we can use merges in the encoding of traits only. It leads to a language similar to languages with family polymorphism and virtual classes (Ernst, 2001). In such languages, nested composition is limited to nested composition of classes. An alternative option is to keep merges in the source language, but forbid parallel application for such merges. Therefore, the core language would still have parallel application, but the source language merges do not. Potentially, we could try to implement an overloading-like semantics for the source language merges, instead of parallel application.

Furthermore, our current dynamic semantics for parallel applications is simple, but naive. One possible way to optimize parallel application is through specialization. For instance, in the code:

```
fac = new[ExtLang] implExt;
```

we could generate a new trait implementation such as:

```
trait facSpecialized ⇒ {
   lit (x: Double) = {print = x.ToString, eval = x}
   add (x:IEval&IPrint) (y:IEval&IPrint) =
     {
        print = "(" ++ x.print ++ " + " ++ y.print ++ ")",
        eval = x.eval + y. eval
     }
}
```

That is we would just *syntactically* merge all components recursively, eliminating a lot of the overhead induced by nested merges. When specialization can be applied, it could lead to very significant performance improvements. However, such an approach assumes that the source code of the original implementations is known, and we have not looked yet at the technical details of how to implement this. Nevertheless, such a specialization approach is related to *flattening of traits* (Nierstrasz *et al.*, 2006), which has been used to generate new traits/classes from the composition of multiple traits.

Finally, we believe that work on the compilation of virtual classes will shed light on how to optimize parallel application, since virtual classes need to be able to encode code that is similar to the fac and e definitions above. As part of future work on the compilation of languages with the merge operator, we plan to look more closely at existing compilation models for virtual classes, and see whether some of the ideas can be employed to have more efficient forms of parallel application.

8.4 Extensions

There are several avenues for future work. In the setting of disjoint intersection types, an obvious extension is *disjoint polymorphism* (Alpuim *et al.*, 2017), which adds polymorphism into calculi with disjoint intersection types. We discuss two other extensions next.

First-class record labels. Instead of having two constructs for record and record projection as our current system does, we believe it is also possible to employ first-class labels. First-class labels are used in some record calculi with extensible rows (Leijen, 2004). Record $\{l = 1\}$ would have type $\mathsf{Lab}\ l \to \mathsf{Int}$ in the new setting, where the type $\mathsf{Lab}\ l$ is the type of label. Therefore, to project a field against the label l, we would apply it to a term of type $\mathsf{Lab}\ l$. Using first-class labels, the parallel application (defined on Figure 17) could be simplified and unified further. Moreover, the syntactic sort vl would not be needed, as labels would already be values.

Splittable union types. Although our calculus does not support union types, splittable types (defined in Figure 11) have the potential to be extended to union types ($A|B$). For example, assume that the rule:

$$A \to C \lhd (A \mid B) \to C \rhd B \to C$$

is added to the existing definition of splittable types. Combined with rule S-BCD-AND, then the following subtyping statement is derivable:

$$(A \to C) \,\&\, (B \to C) <: (A \mid B) \to C$$

This means that two functions with the same input type can be merged and act as a function. Besides this, we believe that it is also possible to add a dual of the rule S-BCD-AND in the modular BCD subtyping to accomplish more union-related subtyping. We hope to investigate this further in the future.

9 Related work

This section discusses various lines of related work, including calculi with a merge operator, record calculi, and some work on OOP languages.

9.1 Calculi with the merge operator and a direct semantics

Intersection types with a merge operator are a key feature of the Forsythe language of Reynolds (1988). Reynolds (1991) also studied a core calculus with similarities to λ_i. However, merges in Forsythe are restricted and use a syntactic criterion to determine what merges are allowed. A merge is permitted only when the second term is a lambda abstraction or a single field record, which makes the structure of merges always biased. To prevent potential ambiguity, the latter overrides the former when they overlap. If formalized as a tree, the right child of every node is a leaf. The only place for primitive types is the left-most component. Forsythe follows the standard call-by-name small-step reduction, during which types are ignored. The reduction rules deal with merges by continuously checking if the second component can be used in the context (abstractions for application, records for projection). This simple approach, however, is unable to reduce merges when (multiple) primitive types are required. Reynolds (1997) admitted this issue in his later work. We use types to select values from a merge and the disjointness restriction guarantees the determinism. Therefore, the order of a value in a merge is not a deciding factor on whether the value is used.

The calculus $\lambda\&$ proposed by Castagna *et al.* (1995) has a restricted version of the merge operator for functions only. The merge operator is indexed by a list of types of its components. Its operational semantics uses the runtime types of values to select the "best approximate" branch of an overloaded function. $\lambda\&$ requires runtime type checking on values, while in TDOS, all type information is already present in type annotations. Another obvious difference is that λ_i supports merges of any type (not just functions), which are useful for applications other than overloading of functions, including multi-field *extensible records with subtyping* (Oliveira *et al.*, 2016); encodings of *objects* and *traits* (Bi & Oliveira, 2018); *dynamic mixins* (Alpuim *et al.*, 2017); or simple forms of *family polymorphism* (Bi *et al.*, 2018).

Several other calculi with intersection types and overloading of functions have been proposed (Castagna & Xu, 2011; Castagna *et al.*, 2015, 2014), but these calculi do not support a merge operator, and thus avoid the ambiguity problems caused by the construct.

9.2 Calculi with a merge operator and an elaboration semantics

Instead of a direct semantics, many recent works (Dunfield, 2014; Oliveira *et al.*, 2016; Alpuim *et al.*, 2017; Bi *et al.*, 2018, 2019) on intersection types employ an elaboration semantics, translating merges in the source language to products (or pairs) in a target language. With an elaboration semantics the subtyping derivations are coercive (Luo, 1999): they produce coercion functions that explicitly convert terms of one type to another in the target language. This idea was first proposed by Dunfield (2014), where she shows how to elaborate a calculus with intersection and union types and a merge operator to a standard call-by-value lambda calculus with products and sums. Dunfield also proposed a direct semantics, which served as inspiration for our work. However, her direct semantics is non-deterministic and lacks subject reduction (as discussed in detail in Section 3.1). Unlike Forsythe and $\lambda\&$, Dunfield's calculus has unrestricted merges and allows a merge to work as an argument. Her calculus is flexible and expressive and can deal with several programs that are not allowed in Forsythe and $\lambda\&$.

To remove the ambiguity issues in Dunfield's work, the original λ_i calculus (Oliveira *et al.*, 2016) forbids overlapping in intersections using the disjointness restriction for all well-formed intersections. In other words, it does not support unrestricted intersections. Because of this restriction, the proof of coherence in the original λ_i is still relatively simple. Likewise, in the following work on the F_i calculus (Alpuim *et al.*, 2017), which extends λ_i with disjoint polymorphism, *all* intersections must be *disjoint*. However, the disjointness restriction causes difficulties because it breaks *stability of type substitutions*. Stability is a desirable property in a polymorphic-type system that ensures that if a polymorphic type is well-formed then any instantiation of that type is also well-formed. Unfortunately, with

	$\lambda_{,,}$	λ_i '16	F_i	NeColus	F_i^+	λ_i	λ_i^+
Disjointness	○	●	●	●	●	●	●
Unrestricted Intersections	●	○	○	●	●	●	●
Determinism or Coherence	No	Coh.	Coh.	Coh.	Coh.	Det.	Det.
Coercion Free	●	○	○	○	○	●	●
Recursion	●	○	○	○	○	●	●
Direct Semantics	●	○	○	○	○	●	●
Subject Reduction	○	-	-	-	-	●	●
BCD Subtyping	○	○	○	●	●	○	●

Fig. 20. Summary of intersection calculi with the merge operator. $\lambda_{,,}$ stands for Dunfield's calculus. Note that λ_i '16 is the original one by Oliveira *et al.* (2016), whereas λ_i is the variant introduced in this paper. (● = yes, ○ = no, - = not applicable)

disjoint intersections only, this property is not true in general. Thus, F_i can only prove a restricted version of stability, which makes its metatheory nontrivial.

Disjointness of all well-formed intersections is only a sufficient (but not necessary) restriction to ensure an unambiguous semantics. The NeColus calculus (Bi *et al.*, 2018) relaxes the restriction without introducing ambiguity. In NeColus $1 : \mathsf{Int} \,\&\, \mathsf{Int}$ is allowed, but the same term is rejected in the original λ_i. NeColus employs the disjointness restriction *only* on merges, but otherwise allows *unrestricted intersections*. Unfortunately, this comes at a cost: it is much harder to prove the coherence of elaboration. Both NeColus and F_i^+ (Bi *et al.*, 2019) (a calculus derived from F_i that allows unrestricted intersections) deal with this problem by establishing coherence using contextual equivalence and a *logical relation* (Tait, 1967; Plotkin, 1973; Statman, 1985) to prove it. The proof method, however, cannot deal with non-terminating programs. In fact, none of the existing calculi with disjoint intersection types supports recursion, which is a severe restriction.

We retain the essence of the power of Dunfield's calculus (modulo the disjointness restrictions to rule out ambiguity), and gain benefits from the direct semantics. Figure 20 summarizes the key differences between our work and prior work, focusing on the most recent work on disjoint intersection types. Note that the row titled "Coercion Free" denotes whether subtyping generates coercions or not. Our calculi are coercion free, while all other calculi based on an elaboration semantics employ coercive subtyping.

9.3 Record calculi with record concatenation and subtyping

As we have seen, in calculi with disjoint intersection types and records, the merge operator concatenates records in a symmetric way. However, designing a record concatenation operator, no matter symmetric or asymmetric, is a difficult problem in calculi with subtyping, as identified by Cardelli & Mitchell (1991). In both cases, a record can "hide" some fields via subsumption to bypass the restriction on types. As far as we know, no existing record calculus in the literature supports nested composition. However, there are numerous designs in the literature with concatenation or subtyping or both.

Asymmetric concatenation without subtyping. Record concatenation is used by Wand (1989) to model multiple inheritance. He has a biased operator, which overrides the first term by the second if they have conflicting fields. Wand (1989) makes every record types explicitly state whether a field is present in it or absent with respect to a fixed set of labels, which could be infinite. It is similar to the algorithm implemented by Rémy (1989) to type-check records in an ML extension, which keeps track of if a field is absent or not.

Symmetric concatenation without subtyping. Harper & Pierce (1991) design a record calculus with symmetric concatenation. Their system keeps the extension and restriction operators in terms and types and generalizes them to work on two records (or record types). A compatibility check is enforced on types, via the typing of record concatenation and type well-formedness definition. Their type quantification only takes care of negative information. For example, $\Lambda a\#l.a$ stands for any type that does not have a field of name l. Although they avoid subtyping, it is still necessary to reason about type equivalence with the type operations in their calculus.

A similar disjointness constraint for symmetric record concatenation is employed for by the language Ur presented by Chlipala (2010), which has a more comprehensive reasoning on type equivalence. Ur is a dependently typed language with first-class labels, designed for statically typed meta-programming with type inference. It encodes disjoint assertions in guarded types. Ur uses type-level computation, including a type-level map operation, to allow flexible and generic programs to be written using records. The semantics is given by elaboration, and a translation of Ur programs into terms of the Calculus of Inductive Constructions ensures type soundness. Like Harper and Pierce's work, Ur has no subtyping as it introduces ambiguity. The absence of subtyping avoids the "hidden fields" problem.

Record subtyping without concatenation. Cardelli and Mitchell propose to use extension and restriction as primitive operators instead of concatenation. They introduce type operators and negative restrictions in record types, so that in their calculus, via bounded quantification, programmers can declare a polymorphic function that takes any records lacking certain fields. Like merges in λ_i^+, extension in their system is conflict-free, which is ensured by static type-checking. Compared to restriction, we can use type annotations to drop fields in a record in λ_i^+. A difference is that the two operators deal with a record and a field, while our system can handle two records in a merge. As identified by Cardelli & Mitchell (1991), with subtyping, a record can "hide" some fields via subsumption to bypass the restriction on types, which makes it hard to capture the absent fields in record calculi. Their solution is to add type operations that can encode negative information in subtyping. Corresponding to the term operators, they have two type operators of the same name: extension on a type requires that the given field is absent from the type; restriction on a type explicitly excludes the field from it, if it has such a field. For example, $\{\}\backslash l$ stands for an empty record with a restriction that cannot have field l. Any subtype cannot have l either. Besides special subtyping rules, they define type equivalence rules to reason about type operations on records.

Subtyping-constraint-based calculi. Rémy (1995) and following work by Pottier (2000) handle both symmetric and asymmetric concatenation in a constraint-based-type system. To deal with record concatenation, type operators or conditional constraints are used to express two branches: either a field exists or is absent, mirroring the reduction of programs. In subtyping, the type of records are distinguished into two forms: rigid record types and flexible record types. A rigid record type of a term reflects all fields in it. Rigid records have no subtyping, but they can be used in a concatenation with another record. Every rigid record type corresponds to a flexible record type, which has subtypes and supertypes. However, flexible records cannot be used with concatenation. In λ_i and λ_i^+, all records are flexible and they can be used with concatenation. Moreover, the subtyping in their systems is not expressive enough to support nested composition.

Record calculi as extensions of system $F_{<:}$. The $F_{<:\rho}$ calculus proposed by Cardelli (1992) extends System $F_{<:}$ by extensible records and combines row quantification used in the previously discussed Harper & Pierce (1991)'s work with bounded quantification. $F_{<:\rho}$ does not have record concatenation as a primitive operator. Instead, it has row extension and restriction. A translation to $F_{<:}$ is provided. Poll (1997) solves the polymorphic record update problem in System F with a restricted formulation of subtyping: it only supports width-subtyping on record types. It has a record-update operator instead of concatenation. One record-update operation only alters a field in a record. The subtype checking in its typing rule makes sure the record contains that field of the expected type.

The $F_{\#}$ calculus by Zwanenburg (1995) supports intersection types (in its later version (Zwanenburg, 1997) intersection types are eliminated) and record concatenation in a $F_{<:}$-like system. Similar to λ_i^+, multi-field records are obtained by concatenating single-field records, and there is a distributivity rule for records in subtyping as well. They use a "with" construct for record concatenation which is similar to the merge operator. Like rule TYP-MERGE, the typing of "with" introduces intersections, and it has a compatibility pre-condiction for the two terms' types (written as $A\#B$). Only record types or *Top* can be compatible. The concatenation operator is asymmetric. When two concatenated records have the same label, the right one overwrites the left. Correspondingly, two compatible types can have common fields as long as for those shared fields, the right one has a subtype of the left's , e.g. $\{l : \mathsf{Int}\}\#\{l : \mathsf{Int}\} \,\&\, \{l : \mathsf{Char}\}$. In contrast, disjointness is symmetric, and a type (unless it is top-like) cannot be disjoint with its subtypes, to ensure the two sides of a merge coexist safely. To prevent the issue of subsumption "hiding" fields of different types the compatibility checking, they require explicit annotations on merged records. These annotations are used during elaboration to a target calculus, therefore affecting the program behavior, like in our calculus. The semantics of $F_{\#}$ is given by elaborating into system F with pairs and records. In this sense, it predates Dunfield's work. Concatenated records are translated into pairs, where a special "overwriter" function, generated by the compatibility derivation, is applied to update the overlapped fields in the first record by the second one. In the work of Zwanenburg (1995), coherence is left for future work.

9.4 Languages and calculi with type-dependent semantics

Typed operational semantics. Goguen (1994) uses types in his reduction, similarly to typed reduction in λ_i. However, Goguen's typed operational semantics is designed for studying meta-theoretic properties, especially strong normalization, and is not aimed to describe type-dependent semantics. Unlike TDOS, in typed operational semantics, the reduction process does not use the additional type information to guide reduction. Instead, the combination of well-typedness and computation provides inversion principles for proving various metatheoretical properties. Typed operational semantics has been applied to several systems. These include *simply typed lambda calculi* (Goguen, 1995), calculi with *dependent types* (Goguen, 1994; Feng & Luo, 2009) and *higher-order subtyping* (Compagnoni & Goguen, 2003). Note that the semantics of these systems does not depend on typing, and the untyped (type-erased) reduction relations are still presented to describe how to evaluate programs.

Type classes. (Wadler & Blott, 1989; Kaes, 1988) are an approach to parametric overloading used in languages like Haskell. The commonly adopted compilation strategy for it is the dictionary passing style elaboration (Wadler & Blott, 1989; Hall *et al.*, 1996; Chakravarty *et al.*, 2005a,b). Other mechanisms inspired by type classes, such as Scala's *implicits* (Oliveira *et al.*, 2010), Agda's *instance arguments* (Devriese & Piessens, 2011) or Ocaml's *modular implicits* (White *et al.*, 2014) have an elaboration semantics as well. In one of the pioneering works of type classes, Kaes (1988) gives two formulations for a direct operational semantics. One of them decides the concrete type of the instance of overloaded functions at *runtime*, by analyzing all arguments after evaluating them. In both Kaes' work and the following work by Odersky *et al.* (1995), the runtime semantics has some restrictions with respect to type classes. For example, overloading on return types (needed, for example for the *read* function in Haskell) is not supported. Interestingly, the semantics of λ_i allows overloading on return types, which is used whenever two functions coexist on a merge. For a detailed example, please refer to Section 7.2.

Gradual typing. Siek & Taha (2006) have become popular over the last few years. Gradual typing is another example of a type-dependent mechanism, since the success or not of an (implicit) cast may depend on the particular type used for the implicit cast. Thus, the semantics of a gradually typed language is type-dependent. Like other type-dependent mechanisms, the semantics of gradually typed source languages is usually given by a (type-dependent) elaboration semantics into a cast calculus, such as the blame calculus of Wadler & Findler (2009) or the threesome calculus of Siek & Wadler (2010).

Multiple dispatching. Clifton *et al.* (2000), Chambers & Chen (1999), Muschevici *et al.* (2008), Park *et al.* (2019) generalize object-oriented dynamic dispatch to determine the overloaded method to invoke based on the runtime type of all its arguments. Similar to TDOS, much of the type information is recovered from type annotations in multiple dispatching mechanisms, but, unlike TDOS, they only use input types to determine the semantics.

9.5 Dealing with conflicts in OOP

There is a rich literature in OOP with various solutions for dealing with (method) conflicts. As we have mentioned throughout the article, the approach to deal with conflicts by employing disjointness and a symmetric merge operator is closely related to the trait model (Schärli *et al.*, 2003) in OOP. In the trait model, the idea is that the composition of traits should only be accepted if there are no conflicts: i.e. conflicts should result in errors. To resolve errors the trait model typically allows operations for renaming and/or removing methods from an inherited trait. We can resolve conflicts in the same way, using subtyping to hide (remove) some values in a merge. Bi & Oliveira (2018) have shown how to build a source language with first-class traits on top of a calculus with a symmetric merge operator. An alternative approach would be to have an asymmetric merge operator, such as the one from Dunfield (2014). In such a case, conflicts would be automatically resolved by simply employing the order of composition. Such a semantics is closely related to the traditional semantics for mixins (Bracha & Cook, 1990; Flatt *et al*., 1998). However, this approach is prone to subtle errors arising from the implicit overriding of values/methods triggered by the automatic (and implicit) resolution of conflicts.

Other approaches to conflicts in class-based languages. Several other approaches to deal with conflicts have been studied in the context of OOP. C++ supports a very expressive model of multiple inheritance that accepts two classes *A* and *B* with conflicting method implementations to be inherited by a third class *C*. To deal with method invocations that have multiple conflicting implementations, programmers in C++ can upcast *C* to either *A* or *B* and then the ambiguity is removed because the static type (*A* or *B*) is used to decide which of the methods to invoke. A simple formal model of such a mechanism in C++ (and some extensions) was described by Wang *et al*. (2018). A closely related approach was proposed by Flatt *et al*. (1998) in the context of mixins for a Java-like setting. The idea there is also that if a class inherits from multiple mixins with conflicting methods, then invocations of conflicted methods can still be disambiguated by first upcasting the class to the mixin with the method implementation that the programmer wishes to invoke. This approach is more flexible than the traditional-biased approach of mixin inheritance that employs the order of mixins in composition to automatically override methods with conflicts. Furthermore, it avoids the issues of unintentional errors due to accidental overriding of methods. In such approaches, *nominal types* play an important role, since the nominal types enable distinguishing possible overlapping (or even equal) structural types. Our work is done in the context of languages with structural types and thus it is closer to the work on record calculi. However, the addition of nominal types would be interesting to investigate and could allow for improved mechanisms for conflict resolution.

Delegation-based languages. More generally, the very dynamic nature of merges is closely related to *delegation*-based OOP languages (Lieberman, 1986; Kniesel, 1999; Fisher & Mitchell, 1995; Ungar & Smith, 1988; Ostermann, 2002; Büchi & Weck, 2000). Delegation, originally introduced by Lieberman (1986), is a form of *dynamic inheritance*. With dynamic inheritance, the inherited implementation is not statically known. This is in contrast to static inheritance (which is widely adopted by mainstream OOP languages), where the inherited class must be known statically: in mainstream OOP languages, when using `class A extends B`, `B` is some concrete (statically known) class. Delegation can itself be further classified into two difference forms: *static delegation* and *dynamic delegation* (Kniesel, 1999). In static delegation, the inherited implementation may be statically unknown, but it cannot be changed after it is "bound" to the object. In dynamic delegation, the inherited implementation is stored in a mutable reference in the object and can be changed at any time. The SEDEL language, for instance, adopts a static delegation model, since the self-reference is immutable.

The Self language (Ungar & Smith, 1988) was the first OOP language to employ delegation. JavaScript has a closely related model and is directly inspired by Self. In Self and JavaScript, the idea is that an object has a property that holds a link to another object, which is called the *prototype*. The prototype object can have a prototype of its own, and so on. This offers a very dynamic model where prototype objects can be changed at runtime, by simply mutating the prototype reference. In other words, Self and JavaScript follow a dynamic delegation approach. Due to their dynamically typed nature, there is no support for statically detecting conflicts in JavaScript or Self.

Research on type systems for delegation-based languages is much less explored compared with class-based languages. Fisher & Mitchell (1995) were among the first to study type systems for delegation. Kniesel (1999) further explored this space and has investigated ways to deal with issues arising from conflicts. In his model, he avoids most accidental overriding conflicts, by adopting a rule that a method `m` in one of the parents can only (implicitly) override another method implementation with the same name in another parent, if the two parents have a common parent class. This way, unrelated methods with the same name cannot be accidentally overridden. The work by Ostermann (2002) adapts the idea of family polymorphism to a delegation-based setting and offers some of the advantages of nested composition that is enabled in λ_i^+. There are a few other statically typed approaches to delegation in the context of OOP programming (Büchi & Weck, 2000; Schaefer *et al.*, 2011). Overall, the main difference is that we are looking at foundational calculi using intersection types and a merge operator in this article, instead of looking at the semantics of high-level OOP languages. One application of the calculi in this paper is indeed to model high-level programming abstractions for OOP languages, but that is not the focus of this work. In contrast, the aforementioned related work on delegation is specifically focused on the semantics of higher level OOP languages and language mechanisms.

9.6 BCD subtyping algorithms

Pierce (1989) developed an algorithm for a form of subtyping close to BCD subtyping using a queue of types. His algorithmic decision procedure $le(\sigma, \overline{\tau}, \tau)$ is equivalent to the declarative judgment $\sigma \leq \overline{\tau} \to \tau$, where $\overline{\tau}$ is the queue, containing known argument types of the right-hand-side function type. When τ is a function type $\tau_1 \to \tau_2$, its argument type τ_1 is added to the queue. When σ is an intersection type $\sigma_1 \,\&\, \sigma_2$, the queue is duplicated on both subbranches in order to reflect the distributivity rule, by distributing the argument types to both components of an intersection type. The rule for function types, top types, and intersection types then takes care of argument types in the queue. Bi *et al.* (2018) adapted Pierce's algorithm to BCD subtyping and extended it with record types without major difficulties.

The decidability of BCD subtyping is shown in several other works (Kurata & Takahashi, 1995; Rehof & Urzyczyn, 2011; Statman, 2015) through manual proofs, and there are also proofs formalized in Coq (Laurent, 2012; Bessai *et al.*, 2016). Bessai *et al.* (2019) developed a fast algorithm verified in Coq. Their algorithm is presented as a relational abstract machine specification, with a long proof due to the mismatch between the styles of the declarative system and algorithmic system. In contrast, our algorithm is defined in a simple relational form, keeping the modularity of existing rules, resulting in a novel, simple, and concise formulation of the metatheory for the algorithm. Of course, the two lines of work have quite distinct goals: while we emphasize the modularity and simplicity of the metatheory, Bessai et al. are interested in a fast algorithm, which justifies the additional complexity in the metatheory of their approach.

Muehlboeck & Tate (2018) developed a framework for subtyping algorithms with intersection and union types. They also showed a variant that supports minimal relevant logic B+, which is a generalized system with intersection and union types, subsuming BCD subtyping. To decide $A <: B$, they rewrite A with (a generalized version of) rule OS-DISTARR as much as possible. In contrast, we split B to make the types match. Siek (2019), inspired by Laurent (2019), proposed a new subtyping system and proved the transitivity lemma directly. Siek keeps the judgment form $A <: B$ (like us), but most subtyping rules require changes and are less modular than our rules. Siek's transitivity proof involves a size measure, while we avoid any size measure by using an alternative relation of types (proper types), which exploits the properties of our splittable-type relation. Both works formalize the transitivity property, as well as soundness and completeness to BCD subtyping in proof assistants.

9.7 Parallel reduction in union-related calculi

Parallel reduction for typing unions. As discussed in Section 7.2, λ_i^+ applies a merge of multiple functions to the input together and reduces every component in the resulting merge simultaneously. Although their system does not have merges, Barbanera *et al.* (1995) also employed a similar evaluation strategy called parallel beta-reduction in a calculus with intersection and union types. They consider beta-redexes rather than components in a merge. Instead of reducing the leftmost beta-redex, all occurrences of the same redex are reduced together. Likewise, their motivation is a typing rule that does not preserve types under conventional beta-reduction. This typing rule eliminates union types. If e has type $A \mid B$, the rule considers two cases: every occurrence of e has type A or has type B, and type-checks the expression twice under the two assumptions. Since beta-reduction can change the syntax form of such subterms, it has to be done in parallel. This kind of reduction strategy was proposed by Knuth (1971) and formalized in Barendregt (1984, Chapter 13), and is known as Gross–Knuth reduction.

10 Conclusion

In this work, we showed how a type-directed operational semantics allows us to address the ambiguity problems of calculi with a merge operator. Therefore, with the TDOS approach, we can answer the question of how to give a direct operational semantics for both the general merge operator in a setting with intersection types, as well as calculi with record concatenation and subtyping. Both of these problems are well known to be challenging in the literature, while at the same time having important practical applications. Compared with the elaboration approach, having a direct semantics avoids the translation process and a target calculus. This simplifies both informal and formal reasoning. For instance, establishing the coherence of elaboration in NeColus (Bi *et al.*, 2018) requires much more sophistication than obtaining the determinism theorem in λ_i^+. Furthermore, the proof method for coherence in NeColus cannot deal with nonterminating programs, whereas dealing with recursion is straightforward in λ_i and λ_i^+. The TDOS approach exploits type annotations to guide reduction. The key component of TDOS is *typed reduction*, which allows values to be further reduced depending on their type.

Acknowledgments

We are grateful to anonymous reviewers and our colleagues Yaoda Zhou, Andong Fan, Yaozhu Sun, Baber Rehman, and Jinhao Tan for their valuable comments that helped improve the presentation and our understanding of this work. We also thank Yaozhu for implementing our TDOS. This work has been sponsored by Hong Kong Research Grants Council projects numbers 17209519 and 17209520.

Conflicts of Interest

None.

References

Ahmed, A. J. (2006) Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Proceedings*. Springer, pp. 69–83.

Alpuim, J., d. S. Oliveira, B. C. & Shi, Z. (2017) Disjoint polymorphism. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, Uppsala, Sweden, April 22-29, 2017, Proceedings. Springer, pp. 1–28.

Barbanera, F., Dezani-Ciancaglini, M. & De'Liguoro, U. (1995) Intersection and union types: Syntax and semantics. *Inform. Comput.* **119**(2), 202–230.

Barendregt, H. (1984) *The Lambda Calculus - Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics.

Barendregt, H., Coppo, M. & Dezani-Ciancaglini, M. (1983) A filter lambda model and the completeness of type assignment. *J. Symb. Logic* **48**(4).

Bessai, J., Dudenhefner, A., Düdder, B. & Rehof, J. (2016) Extracting a formally verified subtyping algorithm for intersection types from ideals and filters. TYPES.

Bessai, J., Rehof, J. & Düdder, B. (2019) Fast verified BCD subtyping. In *Models, Mindsets, Meta: The What, the How, and the Why Not?*, vol. 11200. Lecture Notes in Computer Science. Cham: Springer, pp. 356–371.

Bi, X., d. S. Oliveira, B. C. & Schrijvers, T. (2018) The essence of nested composition. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018*, July 16–21, 2018, Amsterdam, The Netherlands: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 22:1–22:33.

Bi, X. & Oliveira, B. C. d. S. (2018) Typed first-class traits. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 9:1–9:28.

Bi, X., Xie, N., d. S. Oliveira, B. C. & Schrijvers, T. (2019) Distributive disjoint polymorphism for compositional programming. In *Programming Languages and Systems - 28th*

European Symposium on Programming, ESOP 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. Springer, pp. 381–409.

Bracha, G. & Cook, W. R. (1990) Mixin-based inheritance. In *Conference on Object-Oriented Programming Systems, Languages, and Applications/European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*, Ottawa, Canada, October 21-25, 1990, Proceedings. ACM, pp. 303–311.

Büchi, M. & Weck, W. (2000) Generic wrappers. In *European Conference on Object-Oriented Programming (ECOOP)*.

Cardelli, L. (1992) *Extensible Records in a Pure Calculus of Subtyping*. Digital. Systems Research Center.

Cardelli, L. & Mitchell, J. (1991) Operations on records. *Math. Struct. Comput. Sci.* **1**, 3–48.

Cardelli, L. & Wegner, P. (1985) On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.* **17**(4), 471–522.

Castagna, G., Ghelli, G. & Longo, G. (1995) A calculus for overloaded functions with subtyping. *Inf. Comput.* **117**(1), 115–135.

Castagna, G., Nguyen, K., Xu, Z. & Abate, P. (2015) Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, Mumbai, India, January 15-17, 2015. ACM, pp. 289–302.

Castagna, G., Nguyen, K., Xu, Z., Im, H., Lenglet, S. & Padovani, L. (2014) Polymorphic functions with set-theoretic types: part 1: Syntax, semantics, and evaluation. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014*, 2014. ACM, pp. 5–18.

Castagna, G. & Xu, Z. (2011) Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011*, Tokyo, Japan, September 19-21, 2011. ACM. pp. 94–106.

Chakravarty, M. M. T., Keller, G. & Jones, S. L. P. (2005a) Associated type synonyms. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, Tallinn, Estonia, September 26–28, 2005. ACM, pp. 241–253.

Chakravarty, M. M. T., Keller, G., Jones, S. L. P. & Marlow, S. (2005b) Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, Long Beach, California, USA, January 12-14, 2005. ACM, pp. 1–13.

Chambers, C. & Chen, W. (1999) Efficient multiple and predicated dispatching. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 1999)*, Denver, Colorado, USA, November 1-5, 1999. ACM, pp. 238–255.

Chlipala, A. (2010) Ur: statically-typed metaprogramming with type-level record computation. *ACM Sigplan Not.* **45**(6), 122–133.

Clifton, C., Leavens, G. T., Chambers, C. & Millstein, T. D. (2000) Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*, Minneapolis, Minnesota, USA, October 15–19, 2000. ACM, pp. 130–145.

Compagnoni, A. B. & Goguen, H. (2003) Typed operational semantics for higher-order subtyping. *Inf. Comput.* **184**(2), 242–297.

Cook, W. R. & Palsberg, J. (1989) A denotational semantics of inheritance and its correctness. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*.

Coppo, M., Dezani-Ciancaglini, M. & Venneri, B. (1981) Functional characters of solvable terms. *Math. Log. Q.* **27**(2-6), 45–58.

Davies, R. & Pfenning, F. (2000) Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, Montreal, Canada, September 18–21, 2000. ACM. pp. 198–208.

Devriese, D. & Piessens, F. (2011) On the bright side of type classes: Instance arguments in Agda. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011*, Tokyo, Japan, September 19-21, 2011. ACM. pp. 143–155.

Dunfield, J. (2014) Elaborating intersection and union types. *J. Funct. Program.* **24**(2–3), 133–165.

Dunfield, J. & Pfenning, F. (2003) Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003, Warsaw, Poland, Proceedings*. Springer, 250–266.

Ernst, E. (2001) Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, p. 303–326.

Facebook. (2014) Flow. https://flow.org/

Feng, Y. & Luo, Z. (2009) Typed operational semantics for dependent record types. In *Proceedings Types for Proofs and Programs, Revised Selected Papers, TYPES 2009*, Aussois, France, 12-15th May 2009. pp. 30–46.

Fisher, K. & Mitchell, J. (1995) A delegation-based object calculus with subtyping. *Fundamentals of Computation Theory*.

Flatt, M., Krishnamurthi, S. & Felleisen, M. (1998) Classes and mixins. *In POPL 1998, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998*. ACM, pp. 171–183.

Freeman, T. S. & Pfenning, F. (1991) Refinement types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*. ACM, pp. 268–277.

Goguen, H. (1994) *A Typed Operational Semantics for Type Theory*. Ph.D. thesis. University of Edinburgh, UK.

Goguen, H. (1995) Typed operational semantics. In *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA 1995*, 1995, Proceedings. Springer, pp. 186–200.

Hall, C. V., Hammond, K., Jones, S. L. P. & Wadler, P. (1996) Type classes in haskell. *ACM Trans. Program. Lang. Syst.* **18**(2), 109–138.

Harper, R. & Pierce, B. (1991) A record calculus based on symmetric concatenation. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, pp. 131–142.

Huang, X. & Oliveira, B. C. d. S. (2020) A type-directed operational semantics for a calculus with a merge operator. In *34th European Conference on Object-Oriented Programming, ECOOP 2020*.

Kaes, S. (1988) Parametric overloading in polymorphic programming languages. *In ESOP 1988, 2nd European Symposium on Programming, Proceedings*. Springer, pp. 131–144.

Kniesel, G. (1999) Type-safe delegation for runtime component adaptation. In *European Conference on Object-Oriented Programming (ECOOP)*.

Knuth, D. E. (1971) Examples of formal semantics. In *Symposium on Semantics of Algorithmic Languages*. Springer.

Kurata, T. & Takahashi, M. (1995) Decidable properties of intersection type systems. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*. Springer-Verlag, p. 297–311.

Laurent, O. (2012) Intersection types with subtyping by means of cut elimination. *Fundamenta Informaticae*. **121**(1–4), 203–226.

Laurent, O. (2019) Intersection subtyping with constructors. In *Proceedings Twelfth Workshop on Developments in Computational Models and Ninth Workshop on Intersection Types and Related Systems (DCM 2018 and ITRS 2018)*, pp. 73–84.

Leijen, D. (2004) First-class labels for extensible rows. Technical Report UU-CS-2004-51. Dept. of Computer Science, Universiteit Utrecht. UTCS Technical Report.

Lieberman, H. (1986) Using prototypical objects to implement shared behavior in object oriented systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

Luo, Z. (1999) Coercive subtyping. *J. Log. Comput.* **9**(1), 105–130.

Microsoft. (2012) TypeScript. https://www.typescriptlang.org/

Muehlboeck, F. & Tate, R. (2018) Empowering union and intersection types with integrated subtyping. *Proc. ACM Program. Lang.* **2**(OOPSLA), 112:1–112:29.

Muschevici, R., Potanin, A., Tempero, E. D. & Noble, J. (2008) Multiple dispatch in practice. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008*, October 19-23, 2008, Nashville, TN, USA. ACM, pp. 563–582.

Nierstrasz, O., Ducasse, S. & Schärli, N. (2006) Flattening traits. *Journal of Object Technology*. **5**(4), 129–148.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E. & Zenger, M. (2004) An overview of the Scala programming language. Technical report. École Polytechnique Fédérale de Lausanne.

Odersky, M., Wadler, P. & Wehr, M. (1995) A second look at overloading. In *Proceedings of the Seventh International Conference on Functional Programming languages and Computer Architecture, FPCA 1995*. ACM, pp. 135–146.

Oliveira, B. C. d. S., Moors, A. & Odersky, M. (2010) Type classes as objects and implicits. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, October 17–21, 2010, Reno/Tahoe, Nevada, USA. ACM, pp. 341–360.

Oliveira, B. C. d. S., Shi, Z. & Alpuim, J. (2016) Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*, Nara, Japan, September 18–22, 2016. ACM, pp. 364–377.

Ostermann, K. (2002) Dynamically composable collaborations with delegation layers. In *European Conference on Object-Oriented Programming (ECOOP)*.

Palsberg, J. & Zhao, T. (2004) Type inference for record concatenation and subtyping. *Inf. Comput.* **189**(1), 54–86.

Park, G., Hong, J., Jr., G. L. S. & Ryu, S. (2019) Polymorphic symmetric multiple dispatch with variance. *Proc. ACM Program. Lang.* **3**(POPL), 11:1–11:28.

Pierce, B. & Steffen, M. (1997) Higher-order subtyping. *Theor. Comput. Sci*. **176**(1), 235–282.

Pierce, B. C. (1989) A decision procedure for the subtype relation on intersection types with bounded variables. Technical report. School of Computer Science, Carnegie-Mellon University.

Pierce, B. C. (1991) *Programming with Intersection Types and Bounded Polymorphism*. Ph.D. thesis. Carnegie Mellon University.

Pierce, B. C. & Turner, D. N. (1998) Local type inference. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 252–265.

Plotkin, G. (1973) Lambda-definability and logical relations.

Poll, E. (1997) System F with width-subtyping and record updating. In *International Symposium on Theoretical Aspects of Computer Software*. Springer, pp. 439–457.

Pottier, F. (2000) A 3-part type inference engine. In *European Symposium on Programming*. Springer, pp. 320–335.

Pottinger, G. (1980) A type assignment for the strongly normalizable λ-terms. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 561–577.

RedHat. (2011) Ceylon. `https://ceylon-lang.org/`

Rehof, J. & Urzyczyn, P. (2011) Finite combinatory logic with intersection types. In *International Conference on Typed Lambda Calculi and Applications*.

Rémy, D. (1989) Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 77–88.

Rémy, D. (1995) A case study of typechecking with constrained types: Typing record concatenation. In *Presented at the workshop on Advances in Types for Computer Science at the Newton Institute*, Cambridge, UK.

Reynolds, J. C. (1988) Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159. Carnegie Mellon University.

Reynolds, J. C. (1991) The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software, International Conference TACS 1991*, Sendai, Japan, September 24-27, 1991, Proceedings. Springer, pp. 675–700.

Reynolds, J. C. (1997) Design of the programming language Forsythe. In *ALGOL-Like Languages*. Springer, pp. 173–233.

Schaefer, I., Bettini, L. & Damiani, F. (2011) Compositional type-checking for delta-oriented programming. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development, AOSD 2011*. ACM, pp. 43–56.

Schärli, N., Ducasse, S., Nierstrasz, O. & Black, A. P. (2003) Traits: Composable units of behaviour. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Proceedings*. Springer, pp. 248–274.

Siek, J. G. (2019) Transitivity of subtyping for intersection types. *CoRR*. **abs / 1906.09709**.

Siek, J. G. & Taha, W. (2006) Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*.

Siek, J. G. & Wadler, P. (2010) Threesomes, with and without blame. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, Madrid, Spain, January 17-23, 2010. ACM, pp. 365–376.

Statman, R. (1985) Logical relations and the typed λ-calculus. *Inf. Control.* **65**(2/3), 85–97.

Statman, R. (2015) A finite model property for intersection types. *Electron. Proc. Theor. Comput. Sci*. **177**, 1–9.

Tait, W. W. (1967) Intensional interpretations of functionals of finite type I. *J. Symb. Log.* **32**(2), 198–212.

Ungar, D. & Smith, R. B. (1988) Self: the power of simplicity (object-oriented language). In *Thirty-Third IEEE Computer Society International Conference, Digest of Papers*.

Wadler, P. (1998) The expression problem. *Posted on the Java Genericity Mailing List*.

Wadler, P. & Blott, S. (1989) How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, USA, January 11–13, 1989. ACM Press, pp. 60–76.

Wadler, P. & Findler, R. B. (2009) Well-typed programs can't be blamed. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, York, UK, March 22–29, 2009. Proceedings. Springer, pp. 1–16.

Wand, M. (1989) Type inference for record concatenation and multiple inheritance. Proceedings. Fourth Annual Symposium on Logic in Computer Science. pp. 92–97.

Wang, Y., Zhang, H., d. S. Oliveira, B. C. & Servetto, M. (2018) FHJ: A formal model for hierarchical dispatching and overriding. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018*, July 16–21, 2018, Amsterdam, The Netherlands.

White, L., Bour, F. & Yallop, J. (2014) Modular implicits. In *Proceedings ML Family/OCaml Users and Developers Workshops*, *ML/OCaml 2014*, Gothenburg, Sweden, September 4–5, 2014, pp. 22–63.

Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inf. Comput.* **115**(1), 38–94.

Xie, N., Oliveira, B. C. d. S., Bi, X. & Schrijvers, T. (2020) Row and bounded polymorphism via disjoint polymorphism. In *34th European Conference on Object-Oriented Programming, ECOOP 2020*.

Zwanenburg, J. (1995) Record concatenation with intersection types.

Zwanenburg, J. (1997) A type system for record concatenation and subtyping. Technical report. Eindhoven University of Technology.