Direct Foundations for Compositional Programming

Andong Fan[1]
Zhejiang University, Hangzhou, China

Xuejing Huang[1]
The University of Hong Kong, China

Han Xu
Peking University, Beijing, China

Yaozhu Sun
The University of Hong Kong, China

Bruno C. d. S. Oliveira
The University of Hong Kong, China

Abstract

The recently proposed CP language adopts Compositional Programming: a new modular programming style that solves challenging problems such as the Expression Problem. CP is implemented on top of a polymorphic core language with disjoint intersection types called F_i^+. The semantics of F_i^+ employs an elaboration to a target language and relies on a sophisticated proof technique to prove the *coherence* of the elaboration. Unfortunately, the proof technique is technically challenging and hard to scale to many common features, including recursion or impredicative polymorphism. Thus, the original formulation of F_i^+ does not support the two later features, which creates a gap between theory and practice, since CP fundamentally relies on them.

This paper presents a new formulation of F_i^+ based on a *type-directed operational semantics* (TDOS). The TDOS approach was recently proposed to model the semantics of languages with disjoint intersection types (but without polymorphism). Our work shows that the TDOS approach can be extended to languages with disjoint polymorphism and model the full F_i^+ calculus. Unlike the elaboration semantics, which gives the semantics to F_i^+ indirectly via a target language, the TDOS approach gives a semantics to F_i^+ directly. With a TDOS, there is no need for a coherence proof. Instead, we can simply prove that the semantics is *deterministic*. The proof of determinism only uses simple reasoning techniques, such as straightforward induction, and is able to handle problematic features such as recursion and impredicative polymorphism. This removes the gap between theory and practice and validates the original proofs of correctness for CP. We formalized the TDOS variant of the F_i^+ calculus and all its proofs in the Coq proof assistant.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Intersection types, disjoint polymorphism, operational semantics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.18

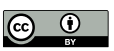
Related Version *Extended Version*: `https://arxiv.org/abs/2205.06150`

Supplementary Material Supplements can be found as follows:
Software (ECOOP 2022 approved artifact): `https://doi.org/10.4230/DARTS.8.2.4`
Software (Coq formalization): `https://github.com/andongfan/CP-Foundations`
Software (Online demo of CP implementation): `https://plground.org`

Funding This research was funded by the University of Hong Kong and Hong Kong Research Grants Council projects number 17209519, 17209520 and 17209821.

Acknowledgements We thank the anonymous reviewers and Wenjia Ye for their helpful comments.

[1] The first two authors contributed equally to this work.



1 Introduction

Compositional Programming [46] is a recently proposed modular programming paradigm. It offers a natural solution to the *Expression Problem* [42] and novel approaches to modular pattern matching and dependency injection. The CP language adopts Compositional Programming. In CP, several new programming language constructs enable Compositional Programming. Of particular interest for this paper, CP has a notion of *typed first-class traits* [5], which are extended in CP to also enable a form of *family polymorphism* [16].

The semantics of CP and its notion of traits is defined via an elaboration to the core calculus F_i^+ [7]: a polymorphic core language with a *merge operator* [34] and *disjoint intersection types* [30]. The elaboration of traits is inspired by Cook's *denotational semantics of inheritance* [12]. In the denotational semantics of inheritance, the key idea is that mechanisms such as classes or traits, which support *self-references* (a.k.a. the `this` keyword in conventional OOP languages), can be modeled via *open recursion*. In other words, the encoding of classes or traits is parametrized by a self-reference. This allows late binding of self-references at the point of instantiation and enables the modification and composition of traits before instantiation. Instantiation happens when **`new`** is used, just as in conventional OOP languages. When **`new`** is used, it essentially closes the recursion by binding the self-reference, which then becomes a recursive reference to the instantiated object. In the denotational semantics of inheritance, **`new`** is just a fixpoint operator.

The semantics of the original formulation of F_i^+ [7] itself is also given by an elaboration into F_{co}, a System F-like language with products. Unlike F_i^+, F_{co} has no subtyping or intersection types, and it has a conventional operational semantics. The main reason for F_i^+ to use elaboration is that F_i^+ has a *type-dependent* semantics: types may affect the runtime behavior of a program. The elaboration semantics for F_i^+ seems like a natural choice, since this is commonly seen in various other type-dependent languages and calculi. For instance, the semantics of type-dependent languages with *type classes* [43], *Scala-style implicits* [29] or *gradual typing* [40] all usually adopt an elaboration approach. In contrast, in the past, more conventional direct formulations using an operational semantics have been avoided for languages with a type-dependent semantics. The appeals of the elaboration semantics are simple type-safety proofs, and the fact that they directly offer an implementation technique over conventional languages without a type-dependent semantics.

There are also important drawbacks when using an elaboration semantics. One of them is simply that more infrastructure is needed for a target language (such as F_{co}) and its associated semantics and metatheory. Moreover, the elaboration semantics is indirect, and to understand the semantics of a program, we must first translate it to the target language (which may be significantly different from the source) and then reason in terms of the target. More importantly, besides type-safety, another property that is often desirable for an elaboration semantics is *coherence* [35]. Many elaboration semantics are non-deterministic: the same source program can elaborate into different target programs. If those different programs have a different semantics, then this is problematic, as it would imply that the source language would have a non-deterministic or ambiguous semantics. Coherence ensures that even if the same program elaborates to different target expressions, the different target expressions are semantically equivalent, eventually evaluating to the same result.

For some languages, including F_i^+, proving coherence is highly non-trivial and hard to scale to common programming language features. For the original F_i^+, the proof of coherence comes at the cost of simple features such as *recursion* and *impredicative polymorphism*. The

proof of coherence for F_i^+ is based on a logical relation called *canonicity* [6]. Together with a notion of contextual equivalence, the two techniques are used to prove coherence. The use of logical relations is a source of complexity in the proof and the reason why recursion and impredicative polymorphism have not been supported. For recursion, in principle, the use of a more sophisticated *step-indexed* logical relation [3] may enable a proof of coherence, at the cost of some additional complexity. However, due to the extra complexity, this was left for future work. For impredicative polymorphism, Bi et al. [7] identified important technical challenges, and it is not known if the proof can be extended with such a feature.

The absence of recursion and impredicative polymorphism creates a gap between theory and practice, since CP fundamentally relies on them. Moreover, the proofs of correctness of CP rely on the assumption that F_i^+ with recursion and impredicative polymorphism would preserve all the properties of F_i^+. Impredicative polymorphism is needed in CP to allow the types of traits with polymorphic methods to be used as type parameters for other polymorphic functions. Recursion is needed in CP because the denotational semantics uses fixpoint operators to instantiate traits. In addition, the fixpoint operators must be *lazy*; otherwise, self-references can easily trigger non-termination. Therefore, a *call-by-name* (CBN) semantics is more natural and also assumed in the CP encoding of traits. However, the semantics of the F_{co} calculus is *call-by-value* (CBV) and, by inheritance, the elaboration semantics of F_i^+ has a CBV semantics as well.

This paper presents a new formulation of F_i^+ based on a *type-directed operational semantics* (TDOS) [20]. The TDOS approach has recently been proposed to model the semantics of languages with disjoint intersection types (but without polymorphism). Although F_i^+ is not a new calculus, we revise its formulation significantly in this paper. Our new formulation of F_i^+ is different from the original one in three aspects. Firstly, the semantics of the original F_i^+ is given by elaborating to F_{co}, while our semantics for F_i^+ is a direct operational semantics. Secondly, our new formulation of F_i^+ supports recursion and impredicative polymorphism. Finally, we employ a call-by-name evaluation strategy.

Our work shows that the TDOS approach can be extended to languages with disjoint polymorphism and model the complete F_i^+ calculus with recursion and impredicative polymorphism. Moreover, there is no need for a coherence proof. Instead, we can simply prove that the semantics is *deterministic*. The proof of determinism uses only simple reasoning techniques, such as straightforward induction, and is able to handle problematic features such as recursion and impredicative polymorphism. Thus, this removes the gap between theory and practice and validates the original proofs of correctness for the CP language. Figure 1 contrasts the differences in terms of proofs and implementation of CP using Zhang et al.'s original work and our own work. We formalized the TDOS variant of the F_i^+ calculus, together with its type-soundness and determinism proof in the Coq proof assistant. Moreover, we have a new implementation of CP based on our new reformulation of F_i^+.

In summary, the contributions of this work are:

- **CBN F_i^+ with recursion and impredicative polymorphism.** This paper presents a CBN variant of F_i^+ extended with recursion and impredicative polymorphism.
- **Determinism and type-soundness for F_i^+ using a TDOS.** We prove the type-soundness and determinism of F_i^+ using a direct TDOS. These proofs validate the proofs of correctness previously presented for CP by Zhang et al. [46].
- **Technical innovations.** Our formulation of F_i^+ has various technical innovations over the original one, including a new formulation of subtyping using splittable types [22] and more flexible term applications.

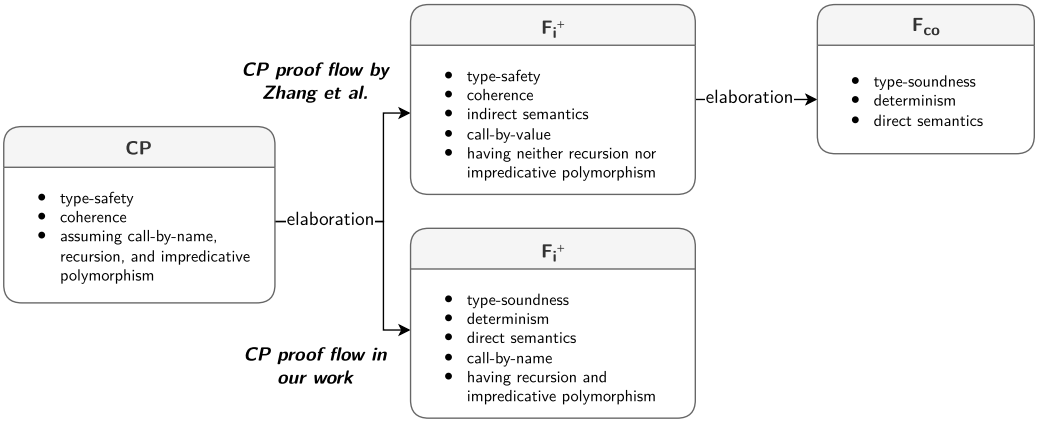


Figure 1 Contrasting the flow of results for CP using the original formulation, and our work.

- **Implementation and Mechanical formalization.** We formalized the TDOS variant of the F_i^+ calculus, together with its type-soundness and determinism proof in the Coq proof assistant. We also have a new implementation of CP built on top of a TDOS formulation of F_i^+ available at `https://plground.org`. The full Coq formalization and the extended version of this paper are available at:

`https://github.com/andongfan/CP-Foundations`

2 Motivations and Technical Innovations

In this section, we introduce Compositional Programming by example and show how CP traits elaborate to F_i^+ expressions. After that, we will discuss the practical issues that motivate us to reformulate F_i^+, as well as technical challenges and innovations.

2.1 Compositional Programming by Example

To demonstrate the capabilities of Compositional Programming, we show how to solve a variant of the *Expression Problem* [42] in the CP language. Our solution is adapted from the original one by Zhang et al. [46]. In this variant, in addition to the usual challenge of extensibility in multiple directions, we also consider the problem of *context evolution* [25, 37], so the interpreter may require different contextual information for different features of the interpreter. The CP language allows a modular solution to both challenges, which also illustrates some key features in Compositional Programming, including *first-class traits* [5], *nested composition* [6], and *disjoint polymorphism* [2].

Examples are based on a simple expression language, and the goal is to perform various operations over it, such as evaluation and free variable bookkeeping. The expression language consists of numbers, addition, variables, and let-bindings. Besides CP code, we also provide analogous Haskell code in the initial examples so that readers can connect them with existing concepts in functional languages.

Compositional interfaces. First, we define the compositional interface for numeric literals and addition. The compositional interface at the top of Figure 2a is similar to Haskell's algebraic data type at the top of Figure 2b. `Exp` is a special kind of type parameter in CP called a *sort*, which serves as the return type of both constructors `Lit` and `Add`. Sorts will

```
type NumSig<Exp> = {
  Lit : Int → Exp;
  Add : Exp → Exp → Exp;
};

type Eval Ctx = { eval : Ctx → Int };
evalNum Ctx = trait implements
    NumSig<Eval Ctx> ⇒ {
  (Lit     n).eval _   = n;
  (Add e1 e2).eval ctx =
     e1.eval ctx + e2.eval ctx;
};
```

(a) CP code.

```
data Exp where
  Lit :: Int → Exp
  Add :: Exp → Exp → Exp

type Eval ctx = ctx → Int

eval :: Exp → Eval ctx
eval (Lit     n) _   = n
eval (Add e1 e2) ctx =
   eval e1 ctx + eval e2 ctx
```

(b) Haskell counterpart.

Figure 2 Initial expression language: numbers and addition.

be instantiated with concrete representations later. Internally, sorts are handled differently from normal type parameters [46]. In accordance with the compositional interface, we can then define how to evaluate the expression language.

Polymorphic contexts. As shown in the middle of Figure 2a, the type `Eval` declares a method `eval` that takes a context and returns an integer. `Ctx` is a type parameter that can be instantiated later, enabling particular traits to assume particular contextual information for the needs of various features. The technique is called *polymorphic contexts* [46] in Compositional Programming.

Compositional traits. The trait `evalNum` in Figure 2a is parametrized by a type parameter `Ctx`. Note that, in CP, type parameters always start with a capital letter, while regular parameters are lowercase. The trait `evalNum` implements the compositional interface `NumSig` by instantiating it with the sort `Eval Ctx`. *Traits* are the basic reusable unit in CP, which are usually type-checked against compositional interfaces. In this trait, we use a lightweight syntax called *method patterns* to define how to evaluate different expressions. Such a definition is analogous to pattern matching in Figure 2b. Since `Lit` and `Add` do not need to be conscious of any information in the context, the type parameter `Ctx` is unconstrained. The only thing that we can do to the polymorphic context is either to ignore it (like in `Lit`) or to pass it to recursive calls (like in `Add`).

More expressions. Adding more constructs to the expression language is awkward in Haskell because algebraic data types are *closed*. However, language components can be modularly declared in CP. Two new constructors, `Let` and `Var`, are declared in the second compositional interface `VarSig`, as shown in Figure 3. Then the two traits implement `VarSig` using method patterns for the new constructors. Since the two new expressions need to inspect or update some information in the context, we expose the appropriate `Env` part to `evalVar`, while the remaining context is kept polymorphic. This is achieved with the *disjointness constraint* [2] `Ctx*Env` in `evalVar`. A disjointness constraint denotes that the type parameter `Ctx` is disjoint to the type `Env`. In other words, types that instantiate `Ctx` cannot overlap with the type `Env`. Also note that the notation `{ ctx with env = ... }` denotes a *polymorphic record update* [9]. In the code for let-expressions, we need to update the environment in the recursive calls to extend it with a new entry for the let-variable.

```
type VarSig<Exp> = {
  Let : String → Exp → Exp → Exp;
  Var : String → Exp;
};

type Env = { env : String → Int };
evalVar (Ctx*Env) = trait implements VarSig<Eval (Env&Ctx)> ⇒ {
  (Let s e1 e2).eval ctx = e2.eval
    { ctx with env = insert s (e1.eval ctx) ctx.env };
  (Var       s).eval ctx = lookup s ctx.env;
};
```

Figure 3 Adding more expressions: variables and let-bindings.

```
type FV = { fv : [String] };
fv = trait implements ExpSig<FV> ⇒ {
  (Lit       n).fv = [];
  (Add   e1 e2).fv = union e1.fv e2.fv;
  (Let s e1 e2).fv = union e1.fv (delete s e2.fv);
  (Var       s).fv = [s];
};

evalWithFV (Ctx*Env) = trait implements ExpSig<FV ⇒ Eval (Env&Ctx)> ⇒ {
  (Lit       n).eval _   = n;
  (Add   e1 e2).eval ctx = e1.eval ctx + e2.eval ctx;
  (Let s e1 e2).eval ctx = if elem s e2.fv
    then e2.eval { ctx with env = insert s (e1.eval ctx) ctx.env }
    else e2.eval ctx;
  (Var       s).eval ctx = lookup s ctx.env;
};
```

Figure 4 Adding more operation: free variable bookkeeping and another version of evaluation.

Intersection types. Independently defined interfaces can be composed using *intersection types*. For example, `ExpSig` below is an intersection of `NumSig` and `VarSig`, containing all of the four constructors:

```
type ExpSig<Exp> = NumSig<Exp> & VarSig<Exp>;
--               = { Lit : ...; Add : ...; Let : ...; Var : ... };
```

More operations. Not only can expressions be modularly extended, but we can easily add more operations. In Figure 4, a new trait `fv` modularly implements a new operation that records free variables in an expression. Here, `union` and `delete` are two library functions for arrays. The modular definition of `fv` is quite natural in functional programming, but it is hard in traditional object-oriented programming. We have to modify the existing class definitions and supplement them with a method. This is typical of the well-known Expression Problem. In summary, we have shown that Compositional Programming can solve both dimensions of this problem: adding expressions and operations.

Dependency injection. Besides the Expression Problem, Figure 4 also shows another significant feature of CP: dependency injection. In `evalWithFV`, a new implementation of evaluation is defined with a dependency on free variables. The method pattern for `Let` will check if `s` appears as a free variable in `e2`. If so, it evaluates `e1` first as usual; otherwise, we do not need to do any computation or update the environment since `s` is not used at all. Note that the compositional interface `ExpSig` is instantiated with two types separated by a fat arrow (⇒) (⇒ was originally denoted by `%` in Zhang et al.'s implementation of CP). `FV` on the left-hand side is the dependency of `evalWithFV`. In other words, the definition of `evalWithFV` depends on another trait that implements `ExpSig<FV>`. The static type checker of CP will check this fact later at the point of trait instantiation. With such dependency injection, we can call `e2.fv` even if `evalWithFV` does not have an implementation of `fv`. In other words, `evalWithFV` depends only on the interface of `fv` (the type `FV`), but not any concrete implementation.

Self-type annotations. Before we show how to perform the new version of the evaluation over the whole expression language, we want to create a repository of expressions for later use. We expect that these expressions are unaware of any concrete operation, so we use a polymorphic `Exp` type to denote some abstract type of expressions. The code that creates the repository of expressions is[1]:

```
repo Exp = trait [self : ExpSig<Exp>] ⇒ {
  num = Add (Lit 4) (Lit 8);
  var = Let "x" (Lit 4) (Let "y" (Lit 8) (Add (Var "x") (Var "y")));
};
```

To make constructors available from the compositional interface, we add a *self-type annotation* to the trait `repo`. The self type annotation `[self : ExpSig<Exp>]` imposes a requirement that the `repo` should finally be merged with some trait implementing `ExpSig<Exp>`. This requirement is also statically enforced by the static type checker of CP. This is the second mechanism in Compositional Programming to modularly inject dependencies.

Nested trait composition. With the language components ready, we can compose them using the merge operator [14], which in the CP language is denoted as a single comma (`,`). First, we show how to compose the old version of the evaluation:

```
exp = new repo @(Eval Env) , evalNum @Env , evalVar @Top;
exp.var.eval { env = empty }  --> 12
```

Since the context has evolved after we add variables, we pass different type arguments to the two traits to make the final context consistent. The final context type is `Env`, so we pass `Env` to `evalNum` and **`Top`** to `evalVar`. Type arguments are prefixed by `@` in CP. A more interesting example is to merge the new version of evaluation with free variable bookkeeping:

```
exp' = new repo @(Eval Env & FV) , evalWithFV @Top , fv;
exp'.var.eval { env = empty }  --> 12
```

After the trait composition, both operations (`eval` and `fv`) are available for expressions that are built with the four constructors (`Lit`, `Add`, `Let`, and `Var`). Note that here `fv` satisfies the dependency of `evalWithFV`. If no implementation of the type `FV` is present in the

[1] In Zhang et al.'s original work [46], the **`new`** operator must be added before every constructor. However, our new implementation will implicitly insert **`new`** (see Section 2.2 for details).

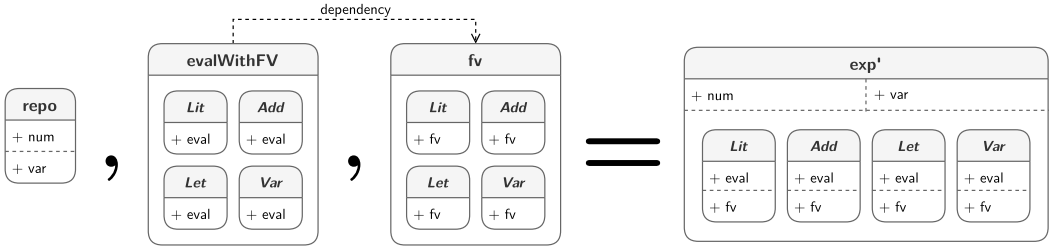


Figure 5 Visualization of nested composition.

composition, there would be a type error, since the requirement for `evalWithFV` would not be satisfied. The composition of the three traits is *nested* because the two methods nested in the four constructors are composed, as visualized in Figure 5. With nested trait composition, the Expression Problem is elegantly solved in Compositional Programming. Moreover, we allow context evolution using a relatively simple way with polymorphic contexts.

Impredicative polymorphism. Another feature of CP is that it allows the creation of objects with polymorphic methods, similar to most OOP languages with generics where classes can contain polymorphic methods (like Java). However, for this to work properly, CP must support *impredicative polymorphism* (the ability to instantiate type parameters with polymorphic types) as System F does. For example, consider:

```
type Poly = { id : forall A. A → A };
idTrait = trait implements Poly ⇒ { id = ΛA. \(x:A) → x };

(new idTrait).id @Poly  -- impredicative
```

While accepted by our variant of CP and F_i^+, such polymorphic instantiations are forbidden in the original formulation of F_i^+.

2.2 Elaborating CP to F_i^+

Under the surface of CP, the foundation for Compositional Programming is the F_i^+ calculus. We present the key features in F_i^+ and take a closer look at the connection between CP and F_i^+ expressions. Here we focus on the elaboration of traits, which are the most important aspect of this paper. We refer curious readers to the work by Zhang et al. [46] for the full formulation of the type-directed elaboration of CP.

Key features of the F_i^+ calculus. F_i^+ is basically a variant of System F [17, 33] extended with intersection types and a merge operator. In the F_i^+ calculus, we denote the merge operator with a double comma $(,,)$ (instead of the single comma notation in CP), following the original notation proposed by Dunfield [14]. The merge operator allows us to introduce terms of intersection types. For example, $1 ,, \mathsf{true}$ is a term of type $\mathsf{Int} \,\&\, \mathsf{Bool}$. Moreover, record concatenation, which is used to encode multi-field traits, is encoded as merges of records in F_i^+. Thus, multi-field records are represented as merges of multiple single-field records. However, to ensure determinism of operational semantics, not all terms can be merged with each other. We impose a disjointness check when typing merges: a merge can only type check when the types of the terms being merged are disjoint, ensuring that every part in a merge can be distinguished by its type. For traits, for example, the disjointness restriction ensures that traits cannot have two fields/methods with the same name `m` and overlapping types, which could otherwise lead to ambiguity when doing method lookup. Here,

we show an example of ambiguity if there is no disjointness check. With intersection types, both $A \,\&\, B \leq A$ and $A \,\&\, B \leq B$ are valid. Therefore, a merge $1\,,\,2$ of type $\mathsf{Int} \,\&\, \mathsf{Int}$ can be typed with Int, but at runtime, two different values of type Int are found. Thus, an expression such as $(1\,,\,2)+1$ could evaluate to either 2 or 3. Since we wish for a deterministic semantics, we use disjointness to prevent such forms of ambiguity. On the other hand, $(1\,,\,\mathsf{true})+1$ type checks because Int and Bool are disjoint, and it evaluates to 2 unambiguously. A disjointness constraint can also be added to a type variable in a System F-style polymorphic type, such as $\forall X * \mathsf{Int}.\, X \,\&\, \mathsf{Int}$. Moreover, to support unrestricted intersection types like $\mathsf{Int} \,\&\, \mathsf{Int}$, the disjointness check is relaxed to *consistency* for certain terms, so that merges with duplications like $1\,,\,1$ are allowed.

Elaborating traits into F_i^+. The elaboration of traits is inspired by Cook's denotational semantics of inheritance [12]. To use a concrete example, we revisit the trait `repo` defined in Section 2.1. Both the creation and instantiation of traits are included in the definition of `repo`:

```
repo Exp = trait [self : ExpSig<Exp>] ⇒ {
  num = Add (Lit 4) (Lit 8);
  var = ...
};
```

The CP code above is elaborated to corresponding F_i^+ code of the form:

```
repo = ΛExp. λ(self : ExpSig<Exp>).
  let $Lit = self.Lit in let $Add = self.Add in
  let $Let = self.Let in let $Var = self.Var in
    { num = fix self:Exp. $Add (fix self:Exp. $Lit 4 self)
                               (fix self:Exp. $Lit 8 self) self } ,,
    { var = ... };
```

The type parameter `Exp` in the `repo` trait is expressed by a System-F-style type lambda ($\Lambda X.\, e$). Note that CP employs a form of syntactic sugar for constructors to allow concise use of constructors and avoid explicit uses of **`new`**. The source code `Add (Lit 4) (Lit 8)` is first expanded into **`new`** `$Add (`**`new`**` $Lit 4) (`**`new`**` $Lit 8)`, which insert **`new`** operators. Next we describe the elaboration process of creating and instantiating traits:

- **Creation of traits**: A **`trait`** is elaborated to a *generator* function whose parameter is a self-reference (like `self` above) and whose body is a record of methods;
- **Instantiation of traits**: The **`new`** construct is used to instantiate a trait. Uses of **`new`** are elaborated to a *fixpoint* which applies the elaborated trait function to a self-reference. In the definition of the field `num` there are three elaborations of **`new`**. For instance, the CP code **`new`** `$Lit 4` corresponds to the F_i^+ code **`fix`** `self:Exp. $Lit 4 self`.

It is clear now that our trait encoding is heavily dependent on recursion, due to the self-references employed by the encoding. However, the original F_i^+ [7] does not support recursion, which reveals a gap between theory and practice.

2.3 The Gap Between Theory and Practice

Our primary motivation to reformulate F_i^+ is to bridge the gap between theory and practice. The original formulation of F_i^+ lacks recursion, impredicative polymorphism and uses the traditional call-by-value (CBV) evaluation strategy. However, the recent work of CP assumes a different variant of F_i^+ that is equipped with fixpoints and the call-by-name (CBN) evaluation. It is worthwhile to probe into the causes of such differences.

Non-triviality of coherence. Recursion is essential for general-purpose computation in programming. More importantly, our encoding of traits requires recursion. For example, **new** e is elaborated to **fix** self. e self. However, adding recursion to the original version of F_i^+ turns out to be highly non-trivial. The original F_i^+ is defined using an elaboration semantics. A fundamental property of F_i^+ is *coherence* [35], which states that the semantics is unambiguous. Coherence is non-trivial due to the presence of the merge operator [14]. To prove coherence, a logical relation, called *canonicity* [7], is used to reason about contextual equivalence in the original work of F_i^+. For example, with contextual equivalence, we can show that the two possible elaborations for the same F_i^+ source expression into F_{co} are contextually equivalent:

$$1 : \mathsf{Int} \,\&\, \mathsf{Int} : \mathsf{Int} \;\leadsto\; \mathsf{fst}\,(1,1) \qquad\qquad 1 : \mathsf{Int} \,\&\, \mathsf{Int} : \mathsf{Int} \;\leadsto\; \mathsf{snd}\,(1,1)$$

Two typing derivations lead to two elaborations in this example, which pick different sides of the merge. However, both elaborated expressions will be reduced to 1 eventually.

Unfortunately, the proof technique for coherence based on logical relations does not immediately scale to recursive programs and programs with impredicative polymorphism. A possible solution, known from the research of logical relations, is to move to a more sophisticated *step-indexed* form of logical relations [1]. However, this requires a major reformulation of the proofs and metatheory of the original F_i^+, and it is not clear whether additional challenges would be present in such an extension. Thus, the lack of the two features in the theory of the original F_i^+ remains a serious limitation since only terminating programs and predicative polymorphism are considered. In other words, we cannot encode traits as presented in Section 2.2 in the original F_i^+. To get around this issue and enable the encoding of traits, Zhang et al. [46] simply assumed an extension of F_i^+ with recursion and their proof of coherence for CP was done under the assumption that the original F_i^+ with recursion was coherent or deterministic.

Our work rectifies this gap in the theory of Compositional Programming and the CP language. We reformulate F_i^+ using a direct type-directed operational semantics [22] that allows recursion and prove that the semantics is deterministic. Thus, our reformulation of F_i^+ can serve as a target language to encode traits and validate the proofs of the elaboration of CP in terms of F_i^+ with recursion. In addition, our approach gives a semantics to F_i^+ directly, instead of relying on an indirect elaboration semantics to a System F-like language.

Evaluation strategies. Most mainstream programming languages use CBV, but CBN is a more natural evaluation strategy for object encodings such as Cook's denotational semantics of inheritance. As stated by Bruce et al. in their work on object encodings [8]:

> *"Although we shall perform conversion steps in whatever order is convenient for the sake of examples, we could just as well impose a call-by-name reduction strategy. (Most of the examples would diverge under a call-by-value strategy. This can be repaired at the cost of some extra lambda abstractions and applications to delay evaluation at appropriate points.)"*

In our elaboration of traits, we adopt a similar approach to object encodings. For example, consider the following CP expression:

```
type A = { l1 : Int; l2 : Int };
new (trait [self : A] ⇒ { l1 = 1; l2 = self.l1 })
```

which is elaborated to the following (slightly simplified) F_i^+ expression:

$$\textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\}$$

The **`trait`** expression is elaborated to a function, and the **`new`** expression turns the function into a fixpoint. Unfortunately, this expression terminates under CBN but diverges under CBV. If evaluated under CBV, the variable *self* will be evaluated repeatedly, despite the fact that only $self.l_1$ is used:

$$\begin{aligned}
& \textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\} \\
\hookrightarrow\ & \{l_1 = 1\}\ ,,\ \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\}).l_1\} \\
\hookrightarrow\ & \{l_1 = 1\}\ ,,\ \{l_2 = (\{l_1 = 1\}\ ,,\ \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\}).l_1\}).l_1\} \\
\hookrightarrow\ & \cdots
\end{aligned}$$

We may tackle the problem of non-termination by wrapping self-references in *thunks*, but CBN provides a simpler and more natural way. In our CBN formulation of F_i^+, $\{l = e\}$ is already a value (instead of $\{l = v\}$), so we do not need to further evaluate e:

$$\begin{aligned}
& \textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\} \\
\hookrightarrow\ & \{l_1 = 1\}\ ,,\ \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\}).l_1\}
\end{aligned}$$

The l_2 field is further evaluated only when a record projection is performed:

$$\begin{aligned}
& (\textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\}).l_2 \\
\hookrightarrow\ & (\{l_1 = 1\}\ ,,\ \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\}).l_1\}).l_2 \\
\hookrightarrow\ & (\textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\}).l_1 \\
\hookrightarrow\ & (\{l_1 = 1\}\ ,,\ \{l_2 = (\textbf{fix}\ self : A.\ \{l_1 = 1\}\ ,,\ \{l_2 = self.l_1\}).l_1\}).l_1 \\
\hookrightarrow\ & 1
\end{aligned}$$

This example illustrates how our new CBN formulation of F_i^+ avoids non-termination of trait instantiation.

2.4 Technical Challenges and Innovations

The main novelty of our reformulation of F_i^+ is the use of a type-directed operational semantics [20] instead of an elaboration semantics. With a TDOS, adding recursion and impredicative polymorphism to our proof of determinism is trivial. Our work is an extension of the λ_i^+ calculus [22] which adapts the TDOS approach. We also follow the subtyping algorithm design in λ_i^+. While λ_i^+ supports BCD-style distributive subtyping [4], the addition of disjoint polymorphism does bring some technical challenges. Moreover, there are some smaller changes to F_i^+ that enable us to type-check more programs and improve the design of the original F_i^+. We will give an overview of the technical challenges and innovations next.

The role of casting. A merge like $1\ ,,\ \mathsf{true}$ has multiple meanings under different types (e.g. Int or Bool). Eventually, we have to extract some components via the elimination of merges, which is a key issue when designing a direct operational semantics for a calculus with the merge operator. A non-deterministic semantics could allow $e_1\ ,,\ e_2 \hookrightarrow e_1$ and $e_1\ ,,\ e_2 \hookrightarrow e_2$ without any constraints, at the cost of losing both type preservation and determinism [14].

To obtain a non-ambiguous and type-safe semantics, we follow the TDOS approach [20]: which uses (up)casts to ensure that values have the right form during reduction. In a TDOS, there is a casting relation, which is used in the reduction rule for annotated values:

$$\frac{v \hookrightarrow_A v'}{v : A \hookrightarrow v'} \textsc{Step-annov}$$

Casting enables us to drop certain parts from a term (e.g., $1\,,,\,\mathsf{true} \hookrightarrow_{\mathsf{Int}} 1$). Very often, it is necessary for us to do so to satisfy the disjointness constraint. Consider a function $\lambda x\!:\!\mathsf{Int}.\, x\,,,\,\mathsf{false}$. For its body to be well-typed, x cannot contain a boolean. Hence, when the function is applied to $1\,,,\,\mathsf{true}$, we cannot directly substitute the argument in. Instead, it is wrapped by (and later cast to) Int to resolve the potential conflict.

$$\begin{array}{ll} & ((\lambda x\!:\!\mathsf{Int}.\, x\,,,\,\mathsf{false}) : \mathsf{Int}\,\&\,\mathsf{Bool} \to \mathsf{Int}\,\&\,\mathsf{Bool})\,(1\,,,\,\mathsf{true}) \\ \hookrightarrow & ((1\,,,\,\mathsf{true}) : \mathsf{Int}\,,,\,\mathsf{false}) : \mathsf{Int}\,\&\,\mathsf{Bool} \\ \hookrightarrow & (1\,,,\,\mathsf{false}) : \mathsf{Int}\,\&\,\mathsf{Bool} \\ \hookrightarrow & 1\,,,\,\mathsf{false} \end{array}$$

TDOS and function annotations. In casting, values in a merge are selected based on type information. In the absence of runtime type-checking, we need to know the type of input value syntactically to match it with the target type. Thus, functions must be accompanied by type annotations. The previous work λ_i^+ [22] defines the syntax of functions like $\lambda x.\, e : A \to C$. While the original argument type A is always kept during reduction, λ_i^+'s casting relation may generate a value that has a proper subtype of the requested type: $\lambda x.\, e : A \to C \hookrightarrow_{B_1 \to B_2} \lambda x.\, e : A \to B_2$. We make casting more precise with a more liberal syntax in F_i^+. We allow bare abstractions $\lambda x\!:\!A.\, e$ while λ_i^+ does not. Our casting relation requires lambdas to be annotated $(\lambda x\!:\!A.\, e) : B$, but the full annotation B does not have to be a function type. For example, $(\lambda x\!:\!\mathsf{Int}.\, x\,,,\,\mathsf{true}) : (\mathsf{Int} \to \mathsf{Int})\,\&\,(\mathsf{Int} \to \mathsf{Bool})$ still acts as a function, and is equivalent to $(\lambda x\!:\!\mathsf{Int}.\, x\,,,\,\mathsf{true}) : \mathsf{Int} \to \mathsf{Int}\,\&\,\mathsf{Bool}$.

Algorithmic subtyping with disjoint polymorphism. F_i^+ extends BCD-style distributive subtyping [4] to disjoint polymorphism. $\forall X * \mathsf{Int}.\, X\,\&\,\mathsf{Int}$ represents the intersection of some type X and Int assuming X is disjoint to Int. Like arrows or records, such universal types distribute over intersections. Hence, $(\forall X * \mathsf{Int}.\, X)\,\&\,(\forall X * \mathsf{Int}.\, \mathsf{Int})$ is a subtype of $\forall X * \mathsf{Int}.\, X\,\&\,\mathsf{Int}$. A well-known challenge in supporting distributivity in the BCD-style subtyping is to obtain an algorithmic formulation of subtyping. There have been many efforts to eliminate the explicit transitivity rule to obtain an algorithmic formulation [26, 31, 39]. Compared with the original F_i^+ [7], we employ a different subtyping algorithm design, using *splittable types* [21]. This approach employs a type-splitting operation $(B \lhd A \rhd C)$ that converts a given type A to an equivalent intersection type $B\,\&\,C$, for example, $A \to B_1\,\&\,B_2$ is split to $A \to B_1$ and $A \to B_2$. The subtyping algorithm uses type splitting whenever an intersection type is expected in the conventional algorithm for subtyping without distributivity, and therefore handles distributivity smoothly and modularly. For space reasons, the novel algorithmic subtyping approach is discussed in the extended version of the paper.

Enhanced subtyping and disjointness with more top-like types. Unlike previous systems with disjoint polymorphism [2, 7], we add a context in subtyping judgments to track the disjointness assumption $X * A$ whenever we open a universal type $\forall X * A.\, B$, similar to the subtyping with F-bounded quantification. The extra information enhances our subtyping: we know a type must be a supertype of Top, if it is disjoint with Bot. This also fixes the following broken property in the original F_i^+, as we now have more types that are *top-like*.

▶ **Definition 2.1** (Disjointness specification). *If A is disjoint with B, any common supertypes they have must be equivalent to* Top.

Keeping this property is necessary for us to obtain a deterministic operational semantics. Meanwhile, we prove our subtyping and disjointness relations are decidable in Coq. Note that in the original F_i^+, the decidability of the two relations was proved manually, although the rest of the proof was mechanized.

3 The F_i^+ Calculus and Its Operational Semantics

This section introduces the F_i^+ calculus, including its static and dynamic semantics.

3.1 Syntax

The syntax of F_i^+ is as follows:

Types	$A, B, C ::= X \mid \mathsf{Int} \mid \mathsf{Top} \mid \mathsf{Bot} \mid A \,\&\, B \mid A \to B \mid \forall X * A.\, B \mid \{l : A\}$
Checkable terms	$p ::= \lambda x : A.\, e \mid \Lambda X.\, e \mid \{l = e\}$
Expressions	$e ::= p \mid x \mid i \mid \top \mid e : A \mid e_1 \,,, e_2 \mid \mathbf{fix}\ x : A.\, e \mid e_1\ e_2 \mid e\ A \mid e.l$
Values	$v ::= p \mid p : A \mid i \mid \top \mid v_1 \,,, v_2$
Term contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Type contexts	$\Delta ::= \cdot \mid \Delta, X * A$

Types. Types include the Top type and the uninhabited type Bot. Intersection types are created with $A \,\&\, B$. Disjoint polymorphism, a key feature of F_i^+, is based on universal types with a disjointness quantifier $\forall X * A.\, B$, expressing that the type variable X is bound inside B and disjoint to type A. $\{l : A\}$ denotes single-field record types, where l is the record label. Multi-field record types are desugared to intersections of single-field ones [36]:

$$\{l_1 : A_1;\ \ldots;\ l_n : A_n\} \triangleq \{l_1 : A_1\} \,\&\, \cdots \,\&\, \{l_n : A_n\}$$

Expressions. As we will explain later with the typing rules, some expressions do not have an inferred type (or principal type), including lambda abstractions, type abstractions, and single-field records. We use metavariable p to represent these expressions, which with optional annotations, are values. Also, note that expressions inside record values do not have to be a value since our calculus employs call-by-name. The merge operator ,, composes two expressions to make a term of an intersection type. The top value $\top$ can be viewed as a merge of zero elements. Fixpoint expressions $\mathbf{fix}\ x : A.\, e$ construct recursive programs. The type annotation A denotes the type of x as well as the whole expression. Like record types, multi-field records are desugared to merges of single-field ones:

$$\{l_1 = e_1;\ \ldots;\ l_n = e_n\} \triangleq \{l_1 = e_1\} \,,, \ldots ,, \{l_n = e_n\}$$

Contexts. We have two contexts: Γ tracks the types of term variables; Δ tracks the disjointness information of type variables, which follows the original design of F_i^+. We use $\Delta \vdash A$, $\vdash \Delta$, and $\Delta \vdash \Gamma$ judgments for the type well-formedness and the context well-formedness (defined in the extended version of the paper). For multiple type well-formedness judgments, we combine them into one, i.e., $\Delta \vdash A, B$ means $\Delta \vdash A$ and $\Delta \vdash B$.

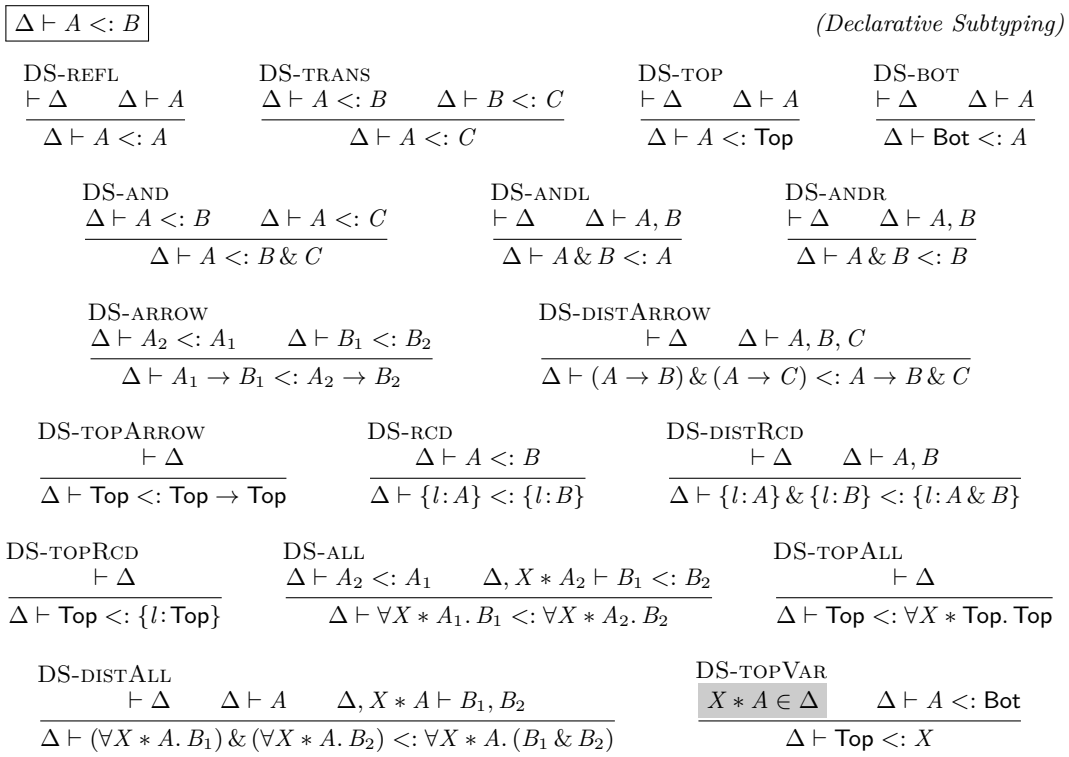


Figure 6 Declarative subtyping rules.

3.2 Subtyping

Figure 6 shows our subtyping relation, which extends BCD-style subtyping [4] with disjoint polymorphism, records, and the bottom type. Compared with the original F_i^+, we add a context to track type variables and their disjointness information. The context not only ensures the well-formedness of types, but is also important to our new rule DS-TOPVAR. An equivalence relation (Definition 3.1) is defined on types that are subtype of each other. These equivalent types can be converted back and forth without loss of information.

▶ **Definition 3.1** (Type equivalence). $\Delta \vdash A \sim B \triangleq \Delta \vdash A <: B$ *and* $\Delta \vdash B <: A$.

For functions (rule DS-ARROW) and disjoint quantifications (rule DS-ALL), subtyping is covariant in positive positions and contravariant in negative positions. The intuition is that type abstractions of the more specific type (subtype) should have a *looser* disjointness constraint for the parameter type. $\forall X * \mathsf{Top}.\, A$ denotes that there is no constraint on X, since Top is disjoint to all types. On the contrary, Bot is the strictest constraint. It is useful in types like $\forall X * \{l : \mathsf{Bot}\}.\, A$ which expresses that X does not contain any informative field of label l [44]. For intersection types, rules DS-ANDL, DS-ANDR, and DS-AND axiomatize that $A \,\&\, B$ is the greatest lower bound of A and B. As a typical characteristic of BCD-style subtyping, type constructors distribute over intersections, including arrows (rule DS-DISTARROW), records (rule DS-DISTRCD) and disjoint quantifications (rule DS-DISTALL).

Another feature of BCD subtyping, which is often overlooked, is the generalization of *top-like types*, i.e. supertypes of Top.

▶ **Definition 3.2** (Specification of top-like types). $\Delta \vdash\rceil A \lceil \triangleq \Delta \vdash A \sim \mathsf{Top}$.

Initially, top-like types include Top and intersections like $\mathsf{Top} \,\&\, \mathsf{Top}$. But the BCD subtyping adds $\mathsf{Top} \to \mathsf{Top}$ to it via rule DS-topArrow, as well as $A \to \mathsf{Top}$ for any type A due to the contravariance of function parameters. Rule DS-topArrow can be viewed as a special case of rule DS-distArrow where intersections are replaced by Top (one can consider it as an intersection of zero components). Like the original F_i^+, we extend this idea to universal types and record types (rules DS-topAll and DS-topRcd).

The most important change is the rule DS-topVar. This rule means that a type variable is top-like if it is disjoint with the bottom type. Every type B is a common supertype of B itself and Bot. If B is disjoint with Bot, then it must be top-like. We proved that subtyping is decidable via an equivalent algorithmic formulation.

The discussion about algorithmic subtyping is in the extended version of the paper.

▶ **Lemma 3.3** (Decidability of subtyping). *$\Delta \vdash A <: B$ is decidable.*

Disjointness. The notion of disjointness (Definition 2.1), defined via subtyping, is used in the original F_i^+, as well as calculi with disjoint intersection types [30]. We proved that our algorithmic definition of disjointness (written as $\Delta \vdash A * B$, in the extended version of the paper) is sound to a specification in terms of top-like types.

▶ **Lemma 3.4** (Disjointness soundness). *If $\Delta \vdash A * B$ then for all type C that $\Delta \vdash A <: C$ and $\Delta \vdash B <: C$ we have $\Delta \vdash\rceil C \lceil$.*

Informally, two disjoint types do not have common supertypes, except for top-like types. This definition is motivated by the desire to prevent ambiguous upcasts on merges. That is, we wish to avoid casts that can extract *different* values of the same type from a merge. Thus in F_i^+ and other calculi with disjoint intersection types, we only allow merges of expressions whose only common supertypes are types that are (equivalent to) the top type. For instance, consider the merge $(1\,,,\mathsf{true})\,,,(2\,,,\text{'c'})$. The first component of the merge $(1\,,,\mathsf{true})$ has type $\mathsf{Int} \,\&\, \mathsf{Bool}$, while the second component $(2\,,,\text{'c'})$ has type $\mathsf{Int} \,\&\, \mathsf{Char}$. This merge is problematic because Int is a supertype of the type of the merge $(\mathsf{Int} \,\&\, \mathsf{Bool}) \,\&\, (\mathsf{Int} \,\&\, \mathsf{Char})$, allowing us to extract two different integers by casting the two terms to Int. Fortunately, our disjointness restriction rejects such merges since the supertype Int is not top-like.

3.3 Bidirectional Typing

The type system of F_i^+ is bidirectional [15], where the subsumption rule is triggered by type annotations. Calculi with a merge operator are incompatible with a general subsumption rule because it cancels disjointness checking. For example, with a general subsumption rule, we can directly use $1\,,,\mathsf{true}$ as a term of type Int since $\mathsf{Int} \,\&\, \mathsf{Bool} <: \mathsf{Int}$. Then, merging $1\,,,\mathsf{true}$ with the term false would type-check since disjointness simply checks whether the static types of merging terms are disjoint, and Int is disjoint with Bool. But now, the merge contains two booleans, which would lead to ambiguity if later we wish to extract a boolean value from the merge. The key issue is that a general subsumption rule loses static type information that is necessary to reject ambiguous merges. A bidirectional type system solves this problem by having a more restricted form of subsumption that only works in the checking mode where the type is provided. A more detailed description of the problem for calculi with the merge operator can be found in Huang et al.'s work [22]. We should also remark that this issue of incompatibility with a general subsumption rule is not unique to calculi with a merge operator. It shows up, for instance, in calculi with *gradual typing* [41] and calculi with *record concatenation* and subtyping [9].

Typing modes $\quad \Leftrightarrow ::= \Leftarrow \mid \Rightarrow$

Pre-values $\quad u ::= i \mid \top \mid e : A \mid u_1 \,,\, u_2$

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

$$\textsc{Typ-top}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \top \Rightarrow \mathsf{Top}} \qquad \textsc{Typ-lit}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \mathsf{Int}} \qquad \textsc{Typ-var}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad x : A \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A}$$

$$\textsc{Typ-abs}\ \frac{\Delta \vdash B_1 <: A \qquad \Delta; \Gamma, x : A \vdash e \Leftarrow B_2}{\Delta; \Gamma \vdash \lambda x : A.\, e \Leftarrow B_1 \to B_2} \qquad \textsc{Typ-tabs}\ \frac{\Delta \vdash \Gamma \qquad \Delta, X * A; \Gamma \vdash e \Leftarrow B}{\Delta; \Gamma \vdash \Lambda X.\, e \Leftarrow \forall X * A.\, B} \qquad \textsc{Typ-rcd}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A}{\Delta; \Gamma \vdash \{l = e\} \Leftarrow \{l : A\}}$$

$$\textsc{Typ-app}\ \frac{\Delta; \Gamma \vdash e_1 \Rightarrow A \qquad A \triangleright B \to C \qquad \Delta; \Gamma \vdash e_2 \Leftarrow B}{\Delta; \Gamma \vdash e_1\, e_2 \Rightarrow C} \qquad \textsc{Typ-tapp}\ \frac{\Delta; \Gamma \vdash e \Rightarrow B \qquad B \triangleright \forall X * C_1.\, C_2 \qquad \Delta \vdash A * C_1}{\Delta; \Gamma \vdash e\, A \Rightarrow C_2[X \mapsto A]} \qquad \textsc{Typ-proj}\ \frac{\Delta; \Gamma \vdash e \Rightarrow A \qquad A \triangleright \{l : C\}}{\Delta; \Gamma \vdash e.l \Rightarrow C}$$

$$\textsc{Typ-merge}\ \frac{\Delta \vdash A * B \qquad \Delta; \Gamma \vdash e_1 \Rightarrow A \qquad \Delta; \Gamma \vdash e_2 \Rightarrow B}{\Delta; \Gamma \vdash e_1 \,,\, e_2 \Rightarrow A \,\&\, B} \qquad \textsc{Typ-mergev}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad u_1 \approx u_2 \qquad \cdot; \cdot \vdash u_1 \Rightarrow A \qquad \cdot; \cdot \vdash u_2 \Rightarrow B}{\Delta; \Gamma \vdash u_1 \,,\, u_2 \Rightarrow A \,\&\, B} \qquad \textsc{Typ-inter}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A \qquad \Delta; \Gamma \vdash e \Leftarrow B}{\Delta; \Gamma \vdash e \Leftarrow A \,\&\, B}$$

$$\textsc{Typ-fix}\ \frac{\Delta; \Gamma, x : A \vdash e \Leftarrow A}{\Delta; \Gamma \vdash \mathbf{fix}\ x : A.\, e \Rightarrow A} \qquad \textsc{Typ-anno}\ \frac{\Delta; \Gamma \vdash e \Leftarrow A}{\Delta; \Gamma \vdash (e : A) \Rightarrow A} \qquad \textsc{Typ-sub}\ \frac{\Delta; \Gamma \vdash e \Rightarrow A \qquad \Delta \vdash A <: B}{\Delta; \Gamma \vdash e \Leftarrow B}$$

Figure 7 Bidirectional typing rules for F_i^+.

Typing. As presented in Figure 7, there are two modes of typing: synthesis ($\Rightarrow$) and checking ($\Leftarrow$). We use $\Leftrightarrow$ as a metavariable for typing modes. $\Delta; \Gamma \vdash e \Leftrightarrow A$ indicates that under type context Δ and term context Γ, the expression e has type A in mode $\Leftrightarrow$. A bidirectional type system directly provides a type-checking algorithm. Δ, Γ, e are all inputs in both modes. Type synthesis generates a *unique* type as the output (also called the inferred type), while type checking takes a type as an input and examines the term.

▶ **Lemma 3.5** (Uniqueness of type synthesis). *If* $\Delta; \Gamma \vdash e \Rightarrow A_1$ *and* $\Delta; \Gamma \vdash e \Rightarrow A_2$ *then* $A_1 = A_2$.

Conversion of typing modes happens in rule Typ-sub. With it, a term with inferred type A can be checked against any B that is a supertype of A. Compared to the original F_i^+, fixpoints are new. They model recursion with a self-reference (x in $\mathbf{fix}\ x : A.\, e$). Other than this, rule Typ-fix is almost the same as rule Typ-anno. It checks the expression e by the annotated type A, with assumption that x has type A in e.

Checking abstractions, type abstractions, and records. To check a function $\lambda x : A.\, e$ against $B_1 \to B_2$ by rule Typ-abs, we track the type of the term variable as *the precise parameter type* A, and check if e can be checked against B_2. B_1 must be a subtype of A

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

$$\text{AD-andArrow}\ \frac{A_1 \rhd B_1 \to C_1 \qquad A_2 \rhd B_2 \to C_2}{A_1 \,\&\, A_2 \rhd B_1 \,\&\, B_2 \to C_1 \,\&\, C_2} \qquad \text{AD-andRcd}\ \frac{A_1 \rhd \{l\!:\!B_1\} \qquad A_2 \rhd \{l\!:\!B_2\}}{A_1 \,\&\, A_2 \rhd \{l\!:\!B_1 \,\&\, B_2\}}$$

$$\text{AD-andAll}\ \frac{A_1 \rhd \forall X * B_1.\ C_1 \qquad A_2 \rhd \forall X * B_2.\ C_2}{A_1 \,\&\, A_2 \rhd \forall X * B_1 \,\&\, B_2.\ (C_1 \,\&\, C_2)} \qquad \text{AD-refl}\ \frac{}{A \rhd A}$$

Figure 8 Applicative distribution rules.

to guarantee the safety of the function application. The type-checking of type abstractions $\Lambda X.\ e$ works by tracking the disjointness relation of the type variable with the context and checking e against the quantified type B. Typing of records works similarly. Additionally, there is a rule Typ-inter, which checks an expression against an intersection type by seperately checking the expression against the composing two types. With this design, we allow $\lambda x\!:\!\mathsf{Int}.\ x\ ,\ ,\ \mathsf{true}$ to be checked against $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$.

Application, record projection, and conversion of applicable types. It is not surprising that a merge can act as a function. But in the original F_i^+, this requires annotations since the expression being applied in an application must have an inferred arrow type. Our design, following the λ_i^+ calculus [22], allows a term of an intersection type to directly apply, as long as the intersection type can be converted into an applicable form. For example, $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$ is converted into $(\mathsf{Int} \,\&\, \mathsf{Int}) \to (\mathsf{Int} \,\&\, \mathsf{Bool})$, which is a supertype of the former. When inferring the type of the application $e_1\ e_2$, rule Typ-app first *converts* the inferred type of e_1 into an arrow form $B \to C$ and then checks the argument e_2 against B. If the check succeeds, the whole expression has inferred type C.

In F_i^+, we have three applicable forms: *arrow types*, *record types*, *universal types*. Like rule Typ-app, the typing of type application and record projection also allows the applied term to have an intersection type, and relies on *applicative distribution* to convert the type.

Applicative distribution $A \rhd B$ (defined in Figure 8) takes type A and generates a supertype B that has an applicable form. The first three rules bring all parts of the input intersection type together. For example, assuming that we apply several merged functions whose types are $A_1 \to B_1$, $A_2 \to B_2$, ..., $A_n \to B_n$, the combined function type is $(A_1 \,\&\, ...\, \&\, A_n) \to (B_1 \,\&\, ...\, \&\, B_n)$. It is equivalent to the input type only when A_1, A_2, ..., and A_n are all equivalent. Essentially, applicative distribution $(A \rhd B)$ is a subset of subtyping $(A <: B)$. The supertype is selected to ensure that when a merge is applied to an argument, every component in the merge is satisfied. Although each one of the three first rules overlaps with the reflexivity rule, for any given type, at most one result has an applicable form.

Since merges are treated as a whole applicable term, programmers can extend functions via a *compositional* approach without modifying the original implementation. It also enables the modular extension of type abstractions and especially records, which play a core role in the trait encoding used in Compositional Programming.

Davies and Pfenning also employ a similar design in their bidirectional type system for refinement intersections [13]. Their type conversion procedure respects subtyping as well. Instead of combining function types, it makes use of $A \,\&\, B <: A$ and $A \,\&\, B <: B$ to enumerate components in intersections and uncover arrows.

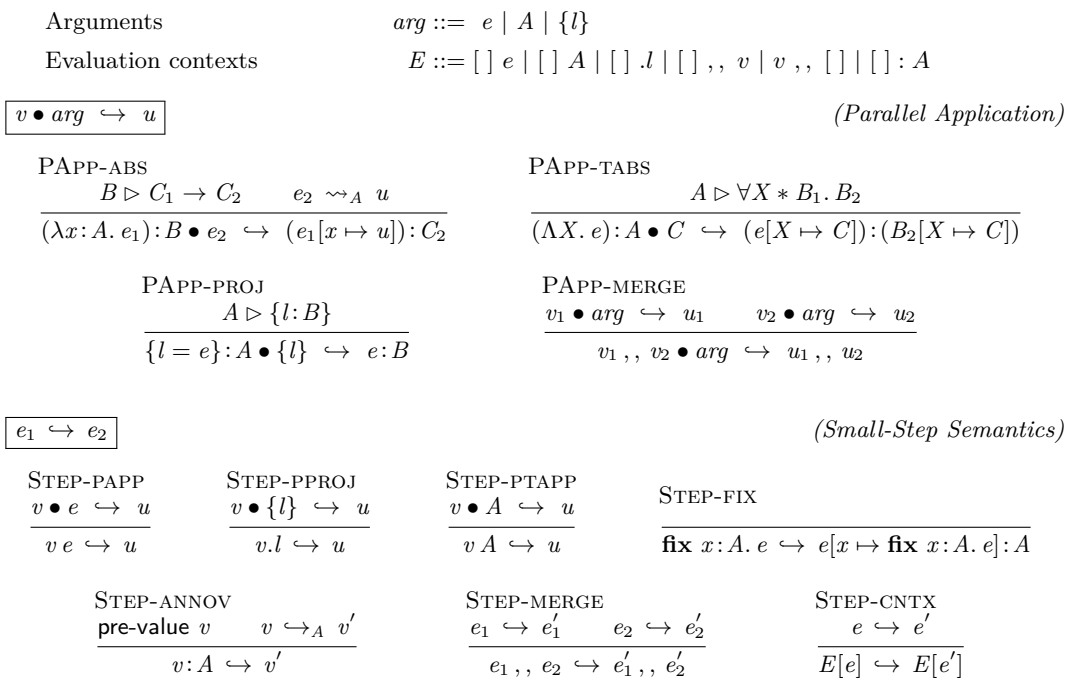


Figure 9 Small-step semantics rules.

Typing merges with disjointness and consistency. Well-typed merges always have inferred types. There are two type synthesis rules for merges, both combining the inferred types of the two parts into an intersection. Typ-merge requires the two subterms to have *disjoint* inferred types, like $1 \,,\, \mathsf{true}$. Typ-mergev relaxes the disjointness constraint to *consistency* checking (written as $u_1 \approx u_2$) to accept overlapping terms like $1 \,,\, 1$. Such duplication is meaningless to users but may appear during evaluation. In fact, rule Typ-mergev is designed for metatheory properties, and not to allow more user-written programs [22]. We will state the formal specification of consistency in Section 4.1 and show how it is involved in the proofs of determinism and type soundness. Informally, consistent merges cause no ambiguity in the runtime. For practical reasons, we only consider *pre-values* (defined at the top of Figure 7) in consistency checking, for which the inferred type can be told directly. The algorithms for disjointness and consistency are presented in the extended version of the paper. In general, disjointness and consistency avoid introducing ambiguity of merges, and enable a deterministic semantics for F_i^+.

3.4 Small-Step Operational Semantics

We specify the *call-by-name* reduction of F_i^+ using a small-step operational semantics in Figure 9. Step-papp, Step-pproj, and Step-ptapp are reduction rules for application and record projection. They trigger *parallel application* (defined in the middle of Figure 9) of merged values to the argument. Rule Step-fix substitutes the fixpoint term variable with the fixpoint expression itself. Note that the result is annotated with A. With the explicit type annotation, the result of reduction preserves the type of the original fixpoint expression.

Through rule Step-annov, values are *cast* to their annotated type. Such values must also be pre-values. This is to filter out checkable terms p including bare abstractions or records without annotations, as $p : A$ is a form of value itself and thus should not step.

A merge of multiple terms may reduce in parallel, as shown in rule Step-merge. Only when one side cannot step, the other side steps alone, as suggested by the evaluation context $E\ ,,\ v$ and $v\ ,,\ E$. Rule Step-cntx is the reduction rule of expressions within an *evaluation context*. Since the rule can be applied repeatedly, we only need evaluation contexts of depth one (shown at the top of Figure 9). Our operational semantics substitutes arguments *wrapped* by type annotations into function bodies, while it forbids the reduction of records since records are values.

Parallel application. Parallel application is at the heart of what we call *nested composition* in CP. It provides the runtime behavior that is necessary to implement nested composition, and it reflects the subtyping distributivity rules at the term level. A merge of functions is treated as one function. The beta reduction of all functions in a merge happens in *parallel* to keep the consistency of merged terms. For type abstractions or records, things are similar. The parallel application handles these applicable merges uniformly via rule PApp-merge. To align record projection with the other two kinds of application, we define *arguments* which abstract expressions, types, and record labels (at the top of Figure 9). In rule PApp-abs, the argument expression is *wrapped* by the function argument type before we substitute it into the function body. Parallel application of type abstractions substitutes the type argument into the body and annotates the body with the substituted disjoint quantified type. Rule PApp-proj projects record fields. Note these three rules have types to annotate the result, since in Typ-abs, Typ-tabs, and Typ-rcd we only type the expression e inside in *checking* mode. With an explicit type annotation, the application preserves types.

Splittable types. Before explaining wrapping or casting, we first introduce *splittable types* [22], which are a key component of our algorithmic formulations of various relations. Ordinary types are the basic units, values of ordinary types can be constructed without the merge operator. As defined at the top of Figure 10, ordinary types do not have intersection types in positive positions. By contrast, splittable types are isomorphic to appropriate intersections. Recall that in BCD-style distributive rules, arrows distribute over intersection, making $\mathsf{Int} \to \mathsf{Int} \,\&\, \mathsf{Bool}$ equivalent to the intersection $(\mathsf{Int} \to \mathsf{Int}) \,\&\, (\mathsf{Int} \to \mathsf{Bool})$. Therefore we say that the former type *splits* into the latter two arrow types. In Figure 10, we extend the type splitting algorithm of λ_i^+ to universal types in correspondence to the distributive subtyping rules (rule DS-distArrow, rule DS-distRcd, and rule DS-distAll). It gives a decision procedure to check whether a type is splittable or ordinary.

▶ **Lemma 3.6** (Type splitting loses no information). *If* $\vdash \Delta$ *and* $\Delta \vdash A$ *and* $B \lhd A \rhd C$ *then* $\Delta \vdash A \sim B \,\&\, C$.

Expression wrapping. Rules for expression wrapping $(e \rightsquigarrow_A u)$ are listed in the middle of Figure 10. Basically, it splits the type A when possible, annotates a duplication of e by each ordinary part of A, and then composes all of them. The only exception is that it never uses top-like types to annotate terms, to avoid ill-typed results like $\{l = 1\} : \mathsf{Int} \to \mathsf{Top}$, but rather generates a normal value whose inferred type is that top-like type, like $(\lambda x : \mathsf{Int}.\ \top) : \mathsf{Int} \to \mathsf{Top}$ (via the *top-like value generating* function $[\![A^\circ]\!]$, defined in the extended version of the paper).

Ordinary types $A^\circ, B^\circ, C^\circ ::= X \mid \mathsf{Int} \mid \mathsf{Top} \mid \mathsf{Bot} \mid A \to B^\circ \mid \forall X * A.\, B^\circ \mid \{l : A^\circ\}$

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

$$\textsc{Sp-arrow}\ \frac{C_1 \lhd B \rhd C_2}{A \to C_1 \lhd A \to B \rhd A \to C_2} \qquad \textsc{Sp-rcd}\ \frac{C_1 \lhd B \rhd C_2}{\{l : C_1\} \lhd \{l : B\} \rhd \{l : C_2\}}$$

$$\textsc{Sp-all}\ \frac{C_1 \lhd B \rhd C_2}{\forall X * A.\, C_1 \lhd \forall X * A.\, B \rhd \forall X * A.\, C_2} \qquad \textsc{Sp-and}\ \frac{}{A \lhd A \,\&\, B \rhd B}$$

$\boxed{e \rightsquigarrow_A u}$ *(Expression Wrapping)*

$$\textsc{EW-top}\ \frac{\cdot \vdash \rceil A^\circ \lceil}{e \rightsquigarrow_{A^\circ} [\![A^\circ]\!]} \qquad \textsc{EW-anno}\ \frac{\cdot \vdash \neg \rceil B^\circ \lceil}{e \rightsquigarrow_{B^\circ} e : B^\circ}$$

$$\textsc{EW-and}\ \frac{B_1 \lhd A \rhd B_2 \qquad e \rightsquigarrow_{B_1} u_1 \qquad e \rightsquigarrow_{B_2} u_2}{e \rightsquigarrow_A u_1 \,,, u_2}$$

$\boxed{v_1 \hookrightarrow_A v_2}$ *(Casting)*

$$\textsc{Cast-int}\ \frac{}{i \hookrightarrow_{\mathsf{Int}} i} \qquad \textsc{Cast-top}\ \frac{\cdot \vdash \rceil A^\circ \lceil}{v \hookrightarrow_{A^\circ} [\![A^\circ]\!]}$$

$$\textsc{Cast-mergel}\ \frac{v_1 \hookrightarrow_{A^\circ} v'}{v_1 \,,, v_2 \hookrightarrow_{A^\circ} v'} \qquad \textsc{Cast-merger}\ \frac{v_2 \hookrightarrow_{A^\circ} v'}{v_1 \,,, v_2 \hookrightarrow_{A^\circ} v'}$$

$$\textsc{Cast-anno}\ \frac{\cdot \vdash \neg \rceil B^\circ \lceil \qquad \cdot \vdash A <: B^\circ}{e : A \hookrightarrow_{B^\circ} e : B^\circ} \qquad \textsc{Cast-and}\ \frac{B_1 \lhd A \rhd B_2 \qquad v \hookrightarrow_{B_1} v_1 \qquad v \hookrightarrow_{B_2} v_2}{v \hookrightarrow_A v_1 \,,, v_2}$$

Figure 10 Type splitting, expression wrapping and value casting rules.

Casting. Casting (shown in Figure 10) is the core of the TDOS, and is triggered by the Step-annov rule. Recalling that only values that are also pre-values will be cast, we can always tell the inferred type of the input value and cast it by any supertype of that inferred type. The definition of casting uses the notion of splittable types. In rule Cast-and, the value is cast under two parts of a splittable type separately, and the results are put together by the merge operator. The following example shows that a merge retains its form when cast under equivalent types.

$$\begin{aligned} & ((\lambda x : \mathsf{Int}.\, x) : \mathsf{Int} \to \mathsf{Int}) \,,, ((\lambda x : \mathsf{Int}.\, \mathsf{true}) : \mathsf{Int} \to \mathsf{Bool}) \\ \hookrightarrow_{(\mathsf{Int}\to\mathsf{Int})\,\&\,(\mathsf{Int}\to\mathsf{Bool})} \quad & ((\lambda x : \mathsf{Int}.\, x) : \mathsf{Int} \to \mathsf{Int}) \,,, ((\lambda x : \mathsf{Int}.\, \mathsf{true}) : \mathsf{Int} \to \mathsf{Bool}) \\ \hookrightarrow_{\mathsf{Int}\to\mathsf{Int}\,\&\,\mathsf{Bool}} \quad & ((\lambda x : \mathsf{Int}.\, x) : \mathsf{Int} \to \mathsf{Int}) \,,, ((\lambda x : \mathsf{Int}.\, \mathsf{true}) : \mathsf{Int} \to \mathsf{Bool}) \end{aligned}$$

In the latter case, the requested type is a *function* type, but the result has an intersection type. This change of type causes a major challenge for type preservation.

For ordinary types, rule Cast-int casts an integer to itself under type Int. Under any ordinary top-like type, the cast result is the output of the *top-like value generator*. The casting of values with annotations works by changing the type annotation to the casting (not top-like) supertype. Rule Cast-mergel and rule Cast-merger make a selection between two merged values. The two rules overlap, but for a well-typed value, the casting result is unique.

Example. We show an example to illustrate the behavior of our semantics:

$$
\begin{array}{ll}
& \text{Let } f := \lambda x{:}\mathsf{Int}\,\&\,\mathsf{Top}.\, x\,,,\,\mathsf{false} \text{ in} \\
& ((f{:}(\mathsf{Int}\,\&\,\mathsf{Top}\to\mathsf{Int})\,\&\,(\mathsf{Int}\,\&\,\mathsf{Top}\to\mathsf{Bool})){:}\mathsf{Int}\,\&\,\mathsf{Bool}\to\mathsf{Int}\,\&\,\mathsf{Bool})\,(1\,,,\,\mathsf{true}) \\
\hookrightarrow^* & \{\text{by rules } \textsc{Step-annov}, \textsc{Cast-and}, \text{ and } \textsc{Cast-anno}\} \\
& (f{:}\mathsf{Int}\,\&\,\mathsf{Bool}\to\mathsf{Int})\,,,\,(f{:}\mathsf{Int}\,\&\,\mathsf{Bool}\to\mathsf{Bool})\,(1\,,,\,\mathsf{true}) \\
\hookrightarrow^* & \{\text{by rules } \textsc{Step-papp}, \text{EW-}\textsc{and}, \text{EW-}\textsc{anno}, \text{ and EW-}\textsc{top}\} \\
& (((1\,,,\,\mathsf{true}){:}\mathsf{Int}\,,,\,\top)\,,,\,\mathsf{false}){:}\mathsf{Int}\,,,\,(((1\,,,\,\mathsf{true}){:}\mathsf{Int}\,,,\,\top)\,,,\,\mathsf{false}){:}\mathsf{Bool} \\
\hookrightarrow^* & \{\text{by rules } \textsc{Step-merge}, \textsc{Step-annov}, \textsc{Cast-int}, \textsc{Cast-mergel}, \text{ and } \textsc{Cast-merger}\} \\
& 1\,,,\,\mathsf{false}
\end{array}
$$

This example shows that a function with a splittable type will be cast to a merge of two copies of itself with different type annotations, i.e., two split results. The application of a merge of functions works by distributing the argument to both functions. Finally, casting selects one side of the merge under the annotated type. From this example, we can see that without the precise parameter annotation of a lambda function (here $\mathsf{Int}\,\&\,\mathsf{Top}$), there is no way to filter the argument $1\,,,\,\mathsf{true}$, causing a conflict.

4 Type Soundness and Determinism

In this section, we show that the operational semantics of F_i^+ is type-sound and deterministic. In F_i^+, determinism also plays a key role in the proof of type soundness. Proving progress is straightforward and is discussed in the extended version of the paper.

4.1 Determinism

A common problem with determinism for calculi with a merge operator is the ambiguity of selection between merged values. In our system, ambiguity is removed by employing disjointness and consistency constraints on merges via typing.

▶ **Definition 4.1** (Consistency specification). $v_1 \approx_{spec} v_2 \triangleq$ *For all type* A *that* $v_1 \hookrightarrow_A v_1'$ *and* $v_2 \hookrightarrow_A v_2'$ *then* $v_1' = v_2'$.

Two values in a merge have no conflicts as long as casting both values under any type leads to the same result. This specification allows v_1 and v_2 to contain identical expressions (may differ in annotations), and terms with disjoint types as such terms can only be cast under top-like types, and the cast result is only decided by that top-like type.

▶ **Lemma 4.2** (Top-like casting is term irrelevant). *If* $\cdot \vdash \rceil A \lceil$ *and* $v_1 \hookrightarrow_A v_1'$ *and* $v_2 \hookrightarrow_A v_2'$ *then* $v_1' = v_2'$.

This is because casting only happens when the given type is a supertype of the cast value's type, and disjoint types only share top-like types as common supertypes (Lemma 3.4).

▶ **Lemma 4.3** (Upcast only). *If* $\cdot;\cdot \vdash v \Rightarrow B$ *and* $v \hookrightarrow_A v'$ *then* $\cdot \vdash B <: A$.

With consistency, casting all well-typed values leads to a unique result. The remaining reduction rules, including expression wrapping and parallel application, are trivially deterministic.

▶ **Lemma 4.4** (Determinism of casting). *If* $\cdot;\cdot \vdash v \Rightarrow B$ *and* $v \hookrightarrow_A v_1$ *and* $v \hookrightarrow_A v_2$, *then* $v_1 = v_2$.

▶ **Theorem 4.5** (Determinism of reduction). *If* $\cdot;\cdot \vdash e \Rightarrow A$ *and* $e \hookrightarrow e_1$ *and* $e \hookrightarrow e_2$ *then* $e_1 = e_2$.

$\boxed{A \lesssim B}$ *(Isomorphic Subtyping)*

$$\text{IS-REFL}\ \frac{}{A \lesssim A} \qquad \text{IS-AND}\ \frac{B_1 \lhd B \rhd B_2 \qquad A_1 \lesssim B_1 \qquad A_2 \lesssim B_2}{A_1 \,\&\, A_2 \lesssim B}$$

Figure 11 Isomorphic subtyping.

4.2 Preservation

Retaining preservation is challenging. When typing merges, we need to satisfy the extra side conditions in rules Typ-merge and Typ-mergev: disjointness and consistency. While the former only depends on types, the latter needs special care.

Consistency. As discussed in Section 3.4, casting may duplicate terms. For example, $1 \hookrightarrow_{\mathsf{Int}\,\&\,\mathsf{Int}} 1\,,\,1$ by rule Cast-and. Rule Typ-mergev is a relaxation of rule Typ-merge to type such merges. We have to ensure any two merged casting results are consistent:

▶ **Lemma 4.6** (Value consistency after casting). *If* $\cdot;\cdot \vdash v \Rightarrow C$ *and* $v \hookrightarrow_A v_1$ *and* $v \hookrightarrow_B v_2$ *then* $v_1 \approx v_2$.

Then we need to make sure that consistency is preserved during reduction.

▶ **Lemma 4.7** (Reduction keeps consistency). *If* $\cdot;\cdot \vdash u_1 \Rightarrow A$ *and* $\cdot;\cdot \vdash u_2 \Rightarrow B$ *and* $u_1 \approx u_2$ *then*

- *if* u_1 *is a value and* $u_2 \hookrightarrow u_2'$ *then* $u_1 \approx u_2'$*;*
- *if* u_2 *is a value and* $u_1 \hookrightarrow u_1'$ *then* $u_1' \approx u_2$*;*
- *if* $u_1 \hookrightarrow u_1'$ *and* $u_2 \hookrightarrow u_2'$ *then* $u_1' \approx u_2'$.

Besides, when parallel application substitutes arguments into merges of applicable terms or projects the wished field, consistency is preserved as well. This requirement enforces us to define consistency not only on values but also on pre-values since the application transforms a value merge into a pre-value merge.

▶ **Lemma 4.8** (Parallel application keeps consistency). *If* $\cdot;\cdot \vdash v_1 \Rightarrow A$ *and* $\cdot;\cdot \vdash v_2 \Rightarrow B$ *and* $v_1 \approx v_2$ *and* $v_1 \bullet arg \hookrightarrow u_1$ *and* $v_2 \bullet arg \hookrightarrow u_2$ *then* $u_1 \approx u_2$ *when*

- *arg is a well-typed expression;*
- *or arg is a label;*
- *or arg is a type* C*; we know* $A \rhd \forall X * A_1.\, A_2$ *and* $B \rhd \forall X * B_1.\, B_2$*; and* $\cdot \vdash C * A_1 \,\&\, B_1$.

Our algorithmic formulation of consistency ($u_1 \approx u_2$, presented in the extended version of the paper) keeps the above properties and is sound to the specification (Definition 4.1). The basic idea is to tear all merges apart and compare every component from u_1 and u_2. They are either the same expression with different annotations or have disjoint types.

▶ **Lemma 4.9** (Consistency soundness). *If* $v_1 \approx v_2$ *then* $v_1 \approx_{spec} v_2$.

Isomorphic subtyping. In F_i^+, types are not always precisely preserved by all reduction steps. Specifically, when we cast a value $v \hookrightarrow_A v'$ (in rule Step-annov) or wrap a term $e \rightsquigarrow_A u$ (in rule PApp-abs), the context expects v' or u to have type A, but this is not always true. In our casting rules shown at the bottom of Figure 10, most values will be reduced to results with the exact type that we want, except for rule Cast-and. The inferred type of the result is always an intersection, which may differ from the original splittable type. To describe the change of types during reduction accurately, we define *isomorphic subtyping* (Figure 11). If $A \lesssim B$, we say A is an isomorphic subtype of B. The following lemma shows that while the two types in an isomorphic subtyping relation may be syntactically different, they are equivalent under an empty type context (i.e. $\cdot \vdash A <: B$ and $\cdot \vdash B <: A$).

▶ **Theorem 4.10** (Isomorphic subtypes are equivalent). *If $A \lesssim B$ then $\cdot \vdash A \sim B$.*

With isomorphic subtyping, we define the preservation property of casting, expression wrapping, and parallel application as follows.

▶ **Lemma 4.11** (Casting preserves typing). *If $\cdot;\cdot \vdash v \Rightarrow A$ and $v \hookrightarrow_B v'$ then there exists a type C such that $\cdot;\cdot \vdash v' \Rightarrow C$ and $C \lesssim B$.*

▶ **Lemma 4.12** (Expression wrapping preserves typing). *If $\cdot;\cdot \vdash e \Leftarrow B$ and $\cdot \vdash B <: A$ and $e \rightsquigarrow_A u$ then there exists a type C such that $\cdot;\cdot \vdash u \Rightarrow C$ and $C \lesssim A$.*

▶ **Lemma 4.13** (Parallel application preserves typing). *If $\cdot;\cdot \vdash v \bullet arg \Rightarrow A$ and $v \bullet arg \hookrightarrow u$ then there exists a type B such that $\cdot;\cdot \vdash u \Rightarrow B$ and $B \lesssim A$.*

Of course, we can prove that the result of casting always has a subtype (or an equivalent type) of the requested type instead of an isomorphic subtype. But it would be insufficient for type preservation of reduction. In summary, if casting or wrapping generates a term of type B when the requested type is A, we need B to satisfy:

- B is a subtype of A because we want a preservation theorem that respects subtyping.
- For any type C, $A * C$ implies $B * C$. This is for the disjointness and consistency checking.
- If A converts into an applicable type C, then B converts into an applicable type too.

Finally, with the lemmas above and isomorphic subtyping, we have the type preservation property of F_i^+. That is, after one or multiple steps of reduction ($\hookrightarrow^*$), the inferred type of the reduced expression is an isomorphic subtype. Therefore, for checked expressions, the initial type-checking always succeeds.

▶ **Theorem 4.14** (Type preservation with isomorphic subtyping). *If $\cdot;\cdot \vdash e \Leftrightarrow A$ and $e \hookrightarrow^* e'$ then there exists a type B such that $\cdot;\cdot \vdash e' \Leftrightarrow B$ and $B \lesssim A$.*

▶ **Corollary 4.15** (Type preservation). *If $\cdot;\cdot \vdash e \Leftrightarrow A$ and $e \hookrightarrow^* e'$ then $\cdot;\cdot \vdash e' \Leftarrow A$.*

5 Related Work

In the following discussion, sometimes we attach the publication year to its calculus name for easy distinction. For instance, F_i^+'19 means the original formulation of F_i^+ by Bi et al. [7].

The merge operator, disjoint intersection types and TDOS. The merge operator for calculi with intersection types was proposed by Reynolds [34]. His original formulation came with significant restrictions to ensure that the semantics is not ambiguous. Castagna [11] showed that a merge operator restricted to functions could model overloading. Dunfield [14]

	$\lambda_{,,}$	λ_i'16	F_i	λ_i^+'18	F_i^+'19	λ_i	λ_i^+	F_i^+
Disjointness	○	●	●	●	●	●	●	●
Unrestricted Intersections	●	○	○	●	●	●	●	●
Determinism / Coherence	No	Coh.	Coh.	Coh.	Coh.	Det.	Det.	Det.
Recursion	●	○	○	○	○	●	●	●
Direct Semantics	●	○	○	○	○	●	●	●
Subject Reduction	○	-	-	-	-	●	●	●
Distributive Subtyping	○	○	○	●	●	○	●	●
Disjoint Polymorphism	○	○	●	○	●	○	○	●
Evaluation Strategy	CBV	CBV	CBV	CBV	CBV	CBV	CBV	CBN

Figure 12 Summary of intersection calculi with the merge operator.
(● = yes, ○ = no, - = not applicable).

proposed a calculus, which we refer to as $\lambda_{,,}$, with an unrestricted merge operator. While powerful, $\lambda_{,,}$ lacked both determinism and subject reduction, though type safety was proved via a *type-directed* elaboration semantics.

To address the ambiguity problems in Dunfield's calculus, Oliveira et al. [30] proposed λ_i'16, which only allows intersections of disjoint types. With that restriction and the use of an elaboration semantics, it was then possible to prove the coherence of λ_i'16, showing that the semantics was not ambiguous. Bi et al. [6] relaxed the disjointness restriction, requiring it only on merges, in a new calculus called λ_i^+'18 (or NeColus). This enabled the use of unrestricted intersections in λ_i^+'18. In addition, they added a more powerful subtyping relation based on the well-known BCD subtyping [4] relation. The new subtyping relation, in turn, enabled nested composition, which is a fundamental feature of Compositional Programming. Unfortunately, both unrestricted intersections and BCD subtyping greatly complicated the coherence proof of λ_i^+'18. To address those issues, Bi et al. turned to an approach based on logical relations and a notion of contextual equivalence.

To address the increasing complexities arising from the elaboration semantics and the coherence proofs, Huang et al. [20, 22] proposed a new approach to model the type-directed semantics of calculi with a merge operator. The type-directed elaboration in λ_i'16 and λ_i^+'18 is replaced by a direct *type-directed operational semantics* (TDOS). In the new TDOS formulations of λ_i and λ_i^+, coercive subtyping is removed since subtyping no longer needs to generate explicit coercion for the elaboration to a target calculus. Instead, runtime implicit (up)casting is used. This is implemented by the casting relation, which was originally called *typed reduction*. Our work adopts TDOS and adds disjoint polymorphism. Disjoint polymorphism is used in Compositional Programming to enable techniques such as polymorphic contexts. We also change the evaluation strategy from call-by-value (CBV) to call-by-name (CBN), motivated by the elaboration of trait instantiation in Compositional Programming. Otherwise, with a CBV semantics, many uses of trait instantiation would diverge.

Calculi with disjoint polymorphism. Disjoint polymorphism was originally introduced in a calculus called F_i by Alpuim et al. [2]. A disjointness constraint is added to universal quantification in order to allow merging components whose type contains type variables. Later, Bi et al. [7] augment it with distributive subtyping in the F_i^+'19 calculus. In addition, the bottom type is added, and unrestricted intersection types are also allowed to fully encode row and bounded polymorphism [44]. Compared to F_i^+'19, our new formulation of F_i^+ adopts a direct semantics, based on a TDOS approach, where simpler proofs of determinism supersede the original proofs of coherence. As a result, recursion and impredicative polymorphism can be easily added. Both features are important to fully support the trait encoding in Compositional Programming. A detailed comparison of calculi with a merge operator, which summarizes our discussion on related work, can be found in Figure 12.

F_i^+ versus $\mathsf{F}_{<:}$. There are quite a few typed object encodings in the literature [8], most of which are based on $\mathsf{F}_{<:}$ [10]. As it is not our goal in this paper to encode full OOP in F_i^+, we will not compare our trait encoding with other object encodings. However, it is still interesting to compare F_i^+ with $\mathsf{F}_{<:}$. Some disadvantages of $\mathsf{F}_{<:}$ have been studied in the literature. It has been shown that, with bounded quantification, the subtyping of $\mathsf{F}_{<:}$ is undecidable [32], and some useful operations like polymorphic record updates [9] are not directly supported. F_i^+ does not have these drawbacks. F_i^+ has decidable subtyping. For $\mathsf{F}_{<:}$ the most common decidable fragment is the so-called kernel $\mathsf{F}_{<:}$ variant [10]. Xie et al. [44] have shown that kernel $\mathsf{F}_{<:}$ is encodable in F_i^+. Therefore the bounded quantification that is present in kernel $\mathsf{F}_{<:}$ can be expressed in F_i^+ as well. In addition, polymorphic record updates can be easily encoded without extra language constructs. For example, concerning a polymorphic record that contains an `x` field among others (`rcd : { x: Int } & R`), the record update `{ rcd with x = 1 }` can be encoded in F_i^+ as `{ x = 1 } ,, (rcd : R)`. In other words, we can rewrite whichever fields we want and then merge the remaining polymorphic part back.

F_i^+ versus row-polymorphic calculi. Row polymorphism provides an alternative way to model extensible record types in System F-like calculi. There are many variants of row-polymorphic calculi in the literature [9, 19, 24, 38]. Among them, the most relevant one with respect to our work is $\lambda^{||}$ by Harper and Pierce [19]. Disjoint quantification has a striking similarity to *constrained quantification* in $\lambda^{||}$. Their *compatibility* constraint plays a similar role to *disjointness* in our system. Furthermore, their merge operator ($||$) can concatenate either two records like our merge operator ($,,$) or two record types like our intersection type operator ($\&$). However, their compatibility constraint and merge operator are only applicable to record types, while we generalize them to arbitrary types. $\lambda^{||}$ has no subtyping and does not allow for distributivity and nested composition either. Disjoint polymorphism also subsumes the form of row polymorphism present in $\lambda^{||}$ as demonstrated by Xie et al. [44]. We refer to Xie et al.'s work for an extended discussion of the relationship between F_i^+ and various other row polymorphic calculi.

Semantics for type-dependent languages. The elaboration semantics approach is commonly used to model the semantics of type-dependent languages and calculi. The appeals of the elaboration semantics are simple type-safety proofs, and the fact that they directly offer an implementation technique over conventional languages without a type-dependent semantics. For instance, the semantics of type-dependent languages with *type classes* [18, 43], *Scala-style implicits* [27, 29] or *gradual typing* [40] all use an elaboration semantics. In contrast, in the past, more conventional direct formulations using an operational semantics have been avoided for languages with a type-dependent semantics. A problem is that the type-dependent semantics introduces complexity in the formulation of an operational semantics since enough type information should be present at runtime and type information needs to be properly propagated. Early work on the semantics of type classes [23, 28], for instance, attempted to employ an operational semantics. However, those approaches had significant practical restrictions in comparison to conventional type classes. The TDOS approach has shown how to overcome important issues when modeling the direct semantics of type-dependent languages. An important advantage of the TDOS approach is that it removes the need for non-trivial coherence proofs. The TDOS approach has also been recently shown to work for modeling the semantics of gradually typed languages directly [45].

6 Conclusion

In this paper, we presented a new formulation of the F_i^+ calculus and showed how it serves as a direct foundation for Compositional Programming. In contrast to the original F_i^+, we adopt a direct semantics based on the TDOS approach and embrace call-by-name evaluation. As a result, the metatheory of F_i^+ is significantly simplified, especially due to the fact that a coherence proof based on logical relations and contextual equivalence is not needed. In addition, our formulation of F_i^+ enables recursion and impredicative polymorphism, validating the original trait encoding by Zhang et al. [46]. We proved the type-soundness and determinism of F_i^+ using the Coq proof assistant. Our research explores further possibilities of the TDOS approach and shows some novel notions that could inspire the design of other calculi with similar features.

Although F_i^+ is already expressive enough to work as a core calculus of the CP language, some useful constructs like type operators are missing. We leave the extension of type-level operations for future work. Another interesting design choice that we want to explore is to lazily evaluate both sides of merges, just like what we have done for record fields, which can help avoid some redundant computation on the unused side of a merge.

References

1 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, 2006.

2 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.

3 Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001.

4 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.

5 Xuan Bi and Bruno C. d. S. Oliveira. Typed First-Class Traits. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

6 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The Essence of Nested Composition. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.

7 Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. Distributive disjoint polymorphism for compositional programming. In *European Symposium on Programming (ESOP)*, 2019.

8 Kim B Bruce, Luca Cardelli, and Benjamin C Pierce. Comparing object encodings. In *International Symposium on Theoretical Aspects of Computer Software*, pages 415–438. Springer, 1997.

9 Luca Cardelli and John C Mitchell. Operations on records. In *International Conference on Mathematical Foundations of Programming Semantics*, 1989.

10 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

11 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Conference on LISP and Functional Programming*, 1992.

12 William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

13 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *International Conference on Functional Programming (ICFP)*, 2000.

14 Jana Dunfield. Elaborating intersection and union types. *Journal of Functional Programming (JFP)*, 24(2-3):133–165, 2014.

15 Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021. doi:10.1145/3450952.

16 Erik Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

17 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

18 Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, March 1996.

19 Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Principles of Programming Languages (POPL)*, 1991.

20 Xuejing Huang and Bruno C. d. S. Oliveira. A type-directed operational semantics for a calculus with a merge operator. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:32, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.26.

21 Xuejing Huang and Bruno C d S Oliveira. Distributing intersection and union types with splits and duality (functional pearl). *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–24, 2021.

22 Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. Taming the merge operator. *Journal of Functional Programming*, 31:e28, 2021. doi:10.1017/S0956796821000186.

23 Stefan Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *ESOP '88*, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

24 Daan Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 5:297–312, 2005.

25 Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 333–343, 1995.

26 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. In *OOPSLA*, 2018.

27 Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: Foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

28 Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 135–146, New York, NY, USA, 1995. Association for Computing Machinery.

29 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *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*, pages 341–360. ACM, 2010. doi:10.1145/1869459.1869489.

30 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*, 2016.

31 Benjamin C Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical report, Carnegie Mellon University, 1989.

32 Benjamin C Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.

33 John C Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

34 John C Reynolds. Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University, 1988.

35 John C. Reynolds. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*, pages 675–700. Springer Berlin Heidelberg, 1991.

36 John C Reynolds. Design of the programming language forsythe. In *ALGOL-like languages*, pages 173–233. Birkhauser Boston Inc., 1997.

37 Tom Schrijvers and Bruno C. d. S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN international conference on functional programming*, pages 32–44, 2011.

38 Mark Shields and Erik Meijer. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 261–275, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/360204.360230.

39 Jeremy G. Siek. Transitivity of subtyping for intersection types. *CoRR*, abs / 1906.09709, 2019. arXiv:1906.09709.

40 Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

41 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2007.

42 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.

43 Philip Wadler and Stephen Blott. 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*, pages 60–76. ACM Press, 1989. doi:10.1145/75277.75283.

44 Ningning Xie, Bruno C d S Oliveira, Xuan Bi, and Tom Schrijvers. Row and bounded polymorphism via disjoint polymorphism. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

45 Wenjia Ye, Bruno C. d. S. Oliveira, and Xuejing Huang. Type-directed operational semantics for gradual typing. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 12:1–12:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

46 Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. Compositional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(3):1–61, 2021.