Distributive Disjoint Polymorphism for Compositional Programming

Xuan Bi[1(✉)], Ningning Xie[1], Bruno C. d. S. Oliveira[1], and Tom Schrijvers[2]

[1] The University of Hong Kong, Hong Kong, China
{xbi,nnxie,bruno}@cs.hku.hk
[2] KU Leuven, Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract. Popular programming techniques such as *shallow embeddings* of Domain Specific Languages (DSLs), *finally tagless* or *object algebras* are built on the principle of *compositionality*. However, existing programming languages only support simple compositional designs well, and have limited support for more sophisticated ones.

This paper presents the F_i^+ calculus, which supports highly modular and compositional designs that improve on existing techniques. These improvements are due to the combination of three features: *disjoint intersection types* with a *merge operator*; *parametric (disjoint) polymorphism*; and *BCD-style distributive subtyping*. The main technical challenge is F_i^+'s proof of coherence. A naive adaptation of ideas used in System F's *parametricity* to *canonicity* (the logical relation used by F_i^+ to prove coherence) results in an ill-founded logical relation. To solve the problem our canonicity relation employs a different technique based on immediate substitutions and a restriction to predicative instantiations. Besides coherence, we show several other important meta-theoretical results, such as type-safety, sound and complete algorithmic subtyping, and decidability of the type system. Remarkably, unlike $\mathsf{F}_{<:}$'s *bounded polymorphism*, disjoint polymorphism in F_i^+ supports decidable type-checking.

1 Introduction

Compositionality is a desirable property in programming designs. Broadly defined, it is the principle that a system should be built by composing smaller subsystems. For instance, in the area of programming languages, compositionality is a key aspect of *denotational semantics* [48,49], where the denotation of a program is constructed from the denotations of its parts. Compositional definitions have many benefits. One is ease of reasoning: since compositional definitions are recursively defined over smaller elements they can typically be reasoned about using induction. Another benefit is that compositional definitions are easy to extend, without modifying previous definitions.

Programming techniques that support compositional definitions include: *shallow embeddings* of Domain Specific Languages (DSLs) [20], *finally tagless* [11], *polymorphic embeddings* [26] or *object algebras* [35]. These techniques

© The Author(s) 2019
L. Caires (Ed.): ESOP 2019, LNCS 11423, pp. 381–409, 2019.
https://doi.org/10.1007/978-3-030-17184-1_14

allow us to create compositional definitions, which are easy to extend without modifications. Moreover, when modeling semantics, both finally tagless and object algebras support *multiple interpretations* (or denotations) of syntax, thus offering a solution to the well-known *Expression Problem* [53]. Because of these benefits these techniques have become popular both in the functional and object-oriented programming communities.

However, programming languages often only support simple compositional designs well, while support for more sophisticated compositional designs is lacking. For instance, once we have multiple interpretations of syntax, we may wish to compose them. Particularly useful is a *merge* combinator, which composes two interpretations [35,37,42] to form a new interpretation that, when executed, returns the results of both interpretations.

The merge combinator can be manually defined in existing programming languages, and be used in combination with techniques such as finally tagless or object algebras. Moreover variants of the merge combinator are useful to model more complex combinations of interpretations. A good example are so-called *dependent* interpretations, where an interpretation does not depend *only* on itself, but also on a different interpretation. These definitions with dependencies are quite common in practice, and, although they are not orthogonal to the interpretation they depend on, we would like to model them (and also mutually dependent interpretations) in a modular and compositional style.

Defining the merge combinator in existing programming languages is verbose and cumbersome, requiring code for every new kind of syntax. Yet, that code is essentially mechanical and ought to be automated. While using advanced meta-programming techniques enables automating the merge combinator to a large extent in existing programming languages [37,42], those techniques have several problems: error messages can be problematic, type-unsafe reflection is needed in some approaches [37] and advanced type-level features are required in others [42]. An alternative to the merge combinator that supports modular multiple interpretations and works in OO languages with support for some form of multiple inheritance and covariant type-refinement of fields has also been recently proposed [55]. While this approach is relatively simple, it still requires a lot of manual boilerplate code for composition of interpretations.

This paper presents a calculus and polymorphic type system with *(disjoint) intersection types* [36], called F_i^+. F_i^+ supports our broader notion of compositional designs, and enables the development of highly modular and reusable programs. F_i^+ has a built-in merge operator and a powerful subtyping relation that are used to automate the composition of multiple (possibly dependent) interpretations. In F_i^+ subtyping is coercive and enables the automatic generation of coercions in a *type-directed* fashion. This process is similar to that of other type-directed code generation mechanisms such as *type classes* [52], which eliminate boilerplate code associated to the *dictionary translation* [52].

F_i^+ continues a line of research on disjoint intersection types. Previous work on *disjoint polymorphism* (the F_i calculus) [2] studied the combination of parametric polymorphism and disjoint intersection types, but its subtyping relation does not support BCD-style distributivity rules [3] and the type system also prevents unrestricted intersections [16]. More recently the $\mathsf{NeColus}$ calculus (or λ_i^+) [5] introduced a system with *disjoint intersection types* and BCD-style distributivity rules, but did not account for parametric polymorphism. F_i^+ is unique in that it combines all three features in a single calculus: *disjoint intersection types* and a *merge operator*; *parametric (disjoint) polymorphism*; and a BCD-style subtyping relation with *distributivity rules*. The three features together allow us to improve upon the finally tagless and object algebra approaches and support advanced compositional designs. Moreover previous work on disjoint intersection types has shown various other applications that are also possible in F_i^+, including: *first-class traits* and *dynamic inheritance* [4], *extensible records* and *dynamic mixins* [2], and *nested composition* and *family polymorphism* [5].

Unfortunately the combination of the three features has non-trivial complications. The main technical challenge (like for most other calculi with disjoint intersection types) is the proof of coherence for F_i^+. Because of the presence of BCD-style distributivity rules, our coherence proof is based on the recent approach employed in λ_i^+ [5], which uses a *heterogeneous* logical relation called *canonicity*. To account for polymorphism, which λ_i^+'s canonicity does not support, we originally wanted to incorporate the relevant parts of System F's logical relation [43]. However, due to a mismatch between the two relations, this did not work. The parametricity relation has been carefully set up with a delayed type substitution to avoid ill-foundedness due to its impredicative polymorphism. Unfortunately, canonicity is a heterogeneous relation and needs to account for cases that cannot be expressed with the delayed substitution setup of the homogeneous parametricity relation. Therefore, to handle those heterogeneous cases, we resorted to immediate substitutions and *predicative instantiations*. We do not believe that predicativity is a severe restriction in practice, since many source languages (e.g., those based on the Hindley-Milner type system like Haskell and OCaml) are themselves predicative and do not require the full generality of an impredicative core language. Should impredicative instantiation be required, we expect that step-indexing [1] can be used to recover well-foundedness, though at the cost of a much more complicated coherence proof.

The formalization and metatheory of F_i^+ are a significant advance over that of F_i. Besides the support for distributive subtyping, F_i^+ removes several restrictions imposed by the syntactic coherence proof in F_i. In particular F_i^+ supports unrestricted intersections, which are forbidden in F_i. Unrestricted intersections enable, for example, encoding certain forms of bounded quantification [39]. Moreover the new proof method is more robust with respect to language extensions. For instance, F_i^+ supports the bottom type without significant complications in the proofs, while it was a challenging open problem in F_i. A final interesting aspect is that F_i^+'s type-checking is decidable. In the design space of languages with polymorphism and subtyping, similar mechanisms have been known to lead to undecidability. Pierce's seminal paper "*Bounded quantification is undecidable*" [40] shows that the contravariant subtyping rule for bounded quantification in $\mathsf{F}_{<:}$ leads to undecidability of subtyping. In F_i^+ the contravariant rule for disjoint quantification retains decidability. Since with unrestricted intersections F_i^+ can express several use cases of bounded quantification, F_i^+ could be an interesting and decidable alternative to $\mathsf{F}_{<:}$.

In summary the contributions of this paper are:

- **The F_i^+ calculus,** which is the first calculus to combine disjoint intersection types, BCD-style distributive subtyping and disjoint polymorphism. We show several meta-theoretical results, such as *type-safety*, *sound and complete algorithmic subtyping*, *coherence* and *decidability* of the type system. F_i^+ includes the *bottom type*, which was considered to be a significant challenge in previous work on disjoint polymorphism [2].
- **An extension of the canonicity relation with polymorphism,** which enables the proof of coherence of F_i^+. We show that the ideas of System F's *parametricity* cannot be ported to F_i^+. To overcome the problem we use a technique based on immediate substitutions and a predicativity restriction.
- **Improved compositional designs:** We show that F_i^+'s combination of features enables improved compositional programming designs and supports automated composition of interpretations in programming techniques like object algebras and finally tagless.
- **Implementation and proofs:** All of the metatheory of this paper, except some manual proofs of decidability, has been mechanically formalized in Coq. Furthermore, F_i^+ is implemented and all code presented in the paper is available. The implementation, Coq proofs and extended version with appendices can be found in https://github.com/bixuanzju/ESOP2019-artifact.

2 Compositional Programming

To demonstrate the compositional properties of F_i^+ we use Gibbons and Wu's shallow embeddings of parallel prefix circuits [20]. By means of several different shallow embeddings, we first illustrate the short-comings of a state-of-the-art compositional approach, popularly known as a *finally tagless* encoding [11], in Haskell. Next we show how parametric polymorphism and distributive intersection types provide a more elegant and compact solution in SEDEL [4], a source language built on top of our F_i^+ calculus.

2.1 A Finally Tagless Encoding in Haskell

The circuit DSL represents networks that map a number of inputs (known as the width) of some type A onto the same number of outputs of the same type. The outputs combine (with repetitions) one or more inputs using a binary associative operator $\oplus : A \times A \rightarrow A$. A particularly interesting class of circuits that can be expressed in the DSL are *parallel prefix circuits*. These represent computations that take $n > 0$ inputs $x_1, \ldots, x_n$ and produce n outputs $y_1, \ldots, y_n$, where $y_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i$.

The DSL features 5 language primitives: two basic circuit constructors and three circuit combinators. These are captured in the Haskell type class `Circuit`:

```
data Width = W { width :: Int }
instance Circuit Width where
  identity n   = W n
  fan n        = W n
  beside c1 c2 =
    W (width c1 + width c2)
  above c1 c2  = c1
  stretch ws c = W (sum ws)
```

(a) Width embedding

```
data Depth = D { depth :: Int }
instance Circuit Depth where
  identity n   = D 0
  fan n        = D 1
  beside c1 c2 =
    D (max (depth c1) (depth c2))
  above c1 c2  = D (depth c1 + depth c2)
  stretch ws c = c
```

(b) Depth embedding

Fig. 1. Two finally tagless embeddings of circuits.

```
class Circuit c where
  identity :: Int → c
  fan      :: Int → c
  beside   :: c → c → c
  above    :: c → c → c
  stretch  :: [Int] → c → c
```

An `identity` circuit with n inputs x_i, has n outputs $y_i = x_i$. A `fan` circuit has n inputs x_i and n outputs y_i, where $y_1 = x_1$ and $y_j = x_1 \oplus x_j\,(j > 1)$. The binary `beside` combinator puts two circuits in parallel; the combined circuit takes the inputs of both circuits to the outputs of both circuits. The binary `above` combinator connects the outputs of the first circuit to the inputs of the second; the width of both circuits has to be same. Finally, `stretch ws c` interleaves the wires of circuit `c` with bundles of additional wires that map their input straight on their output. The `ws` parameter specifies the width of the consecutive bundles; the ith wire of `c` is preceded by a bundle of width $ws_i - 1$.

Basic width and depth embeddings. Figure 1 shows two simple shallow embeddings, which represent a circuit respectively in terms of its width and its depth. The former denotes the number of inputs/outputs of a circuit, while the latter is the maximal number of $\oplus$ operators between any input and output. Both definitions follow the same setup: a new Haskell datatype (`Width`/`Depth`) wraps the primitive result value and provides an instance of the `Circuit` type class that interprets the 5 DSL primitives accordingly. The following code creates a so-called Brent-Kung parallel prefix circuit [9]:

```
e1 :: Width
e1 = above (beside (fan 2) (fan 2))
       (above (stretch [2, 2] (fan 2))
         (beside (beside (identity 1) (fan 2)) (identity 1)))
```

Here `e1` evaluates to `W {width = 4}`. If we want to know the depth of the circuit, we have to change type signature to `Depth`.

Interpreting multiple ways. Fortunately, with the help of polymorphism we can define a type of circuits that support multiple interpretations at once.

```
type DCircuit = forall c. Circuit c ⇒ c
```

This way we can provide a single Brent-Kung parallel prefix circuit definition that can be reused for different interpretations.

```
brentKung :: DCircuit
brentKung = above (beside (fan 2) (fan 2))
              (above (stretch [2, 2] (fan 2))
                (beside (beside (identity 1) (fan 2)) (identity 1)))
```

A type annotation then selects the desired interpretation. For instance, `brentKung :: Width` yields the width and `brentKung :: Depth` the depth.

Composition of embeddings. What is not ideal in the above code is that the same `brentKung` circuit is processed twice, if we want to execute both interpretations. We can do better by processing the circuit only once, computing both interpretations simultaneously. The finally tagless encoding achieves this with a boilerplate instance for tuples of interpretations.

```
instance (Circuit c1, Circuit c2) ⇒ Circuit (c1, c2) where
  identity n    = (identity n, identity n)
  fan n         = (fan n, fan n)
  beside c1 c2  = (beside (fst c1) (fst c2), beside (snd c1) (snd c2))
  above c1 c2   = (above (fst c1) (fst c2), above (snd c1) (snd c2))
  stretch ws c  = (stretch ws (fst c), stretch ws (snd c))
```

Now we can get both embeddings simultaneously as follows:

```
e12 :: (Width, Depth)
e12 = brentKung
```

This evaluates to `(W {width = 4}, D {depth = 2})`.

Composition of dependent interpretations. The composition above is easy because the two embeddings are orthogonal. In contrast, the composition of dependent interpretations is rather cumbersome in the standard finally tagless setup. An example of the latter is the interpretation of circuits as their well-sizedness, which captures whether circuits are well-formed. This interpretation depends on the interpretation of circuits as their width.[1]

```
data WellSized = WS { wS :: Bool, ox :: Width }
instance Circuit WellSized where
 identity n   = WS True (identity n)
 fan n        = WS True (fan n)
 beside c1 c2 = WS (wS c1 && wS c2) (beside (ox c1) (ox c2))
```

[1] Dependent recursion schemes are also known as *zygomorphism* [18] after the ancient Greek word $\zeta\upsilon\gamma o\nu$ for yoke. We have labeled the `Width` field with `ox` because it is pulling the yoke.

```
above c1 c2  = WS (wS c1 && wS c2 && width (ox c1) == width (ox c2))
                  (above (ox c1) (ox c2))
stretch ws c = WS (wS c && length ws==width (ox c)) (stretch ws (ox c))
```

The `WellSized` datatype represents the well-sizedness of a circuit with a Boolean, and also keeps track of the circuit's width. The 5 primitives compute the well-sizedness in terms of both the width and well-sizedness of the subcomponents. What makes the code cumbersome is that it has to explicitly delegate to the `Width` interpretation to collect this additional information.

With the help of a substantially more complicated setup that features a dozen Haskell language extensions, and advanced programming techniques, we can make the explicit delegation implicit (see the appendix). Nevertheless, that approach still requires *a lot of boilerplate* that needs to be repeated for each DSL, as well as explicit projections that need to be written in each interpretation. Another alternative Haskell encoding that also enables multiple dependent interpretations is proposed by Zhang and Oliveira [55], but it does not eliminate the explicit delegation and still requires substantial amounts of boilerplate. A final remark is that adding new primitives (e.g., a "right stretch" `rstretch` combinator [25]) can also be easily achieved [46].

2.2 The SEDEL Encoding

SEDEL is a source language that elaborates to F_i^+, adding a few convenient source level constructs. The SEDEL setup of the circuit DSL is similar to the finally tagless approach. Instead of a `Circuit c` type class, there is a `Circuit[C]` type that gathers the 5 circuit primitives in a record. Like in Haskell, the type parameter `C` expresses that the interpretation of circuits is a parameter.

```
type Circuit[C] = {
  identity : Int → C, fan : Int → C, beside : C → C → C,
  above : C → C → C, stretch : List[Int] → C → C };
```

As a side note if a new constructor (e.g., `rstretch`) is needed, then this is done by means of intersection types (`&` creates an intersection type) in SEDEL:

```
type NCircuit[C] = Circuit[C] & { rstretch : List[Int] → C → C };
```

Figure 2 shows the two basic shallow embeddings for width and depth. In both cases, a named SEDEL definition replaces the corresponding unnamed Haskell type class instance in providing the implementations of the 5 language primitives for a particular interpretation.

The use of the SEDEL embeddings is different from that of their Haskell counterparts. Where Haskell implicitly selects the appropriate type class instance based on the available type information, in SEDEL the programmer explicitly selects the implementation following the style used by object algebras. The following code does this by building a circuit with `l1` (short for `language1`).

```
l1 = language1;
e1 = l1.above (l1.beside (l1.fan 2) (l1.fan 2))
```

```
type Width = { width : Int };
language1 : Circuit[Width] = {
  identity (n : Int) = { width = n },
  fan      (n : Int) = { width = n },
  beside   (c1 : Width) (c2 : Width) = { width = c1.width + c2.width },
  above    (c1 : Width) (c2 : Width) = { width = c1.width },
  stretch  (ws : List[Int]) (c : Width) = { width = sum ws } };
```

```
type Depth = { depth : Int };
language2 : Circuit[Depth] = {
  identity (n : Int) = { depth = 0 },
  fan      (n : Int) = { depth = 1 },
  beside   (c1 : Depth) (c2 : Depth) = { depth = max c1.depth c2.depth},
  above    (c1 : Depth) (c2 : Depth) = { depth = c1.depth + c2.depth},
  stretch  (ws : List[Int]) (c : Depth) = { depth = c.depth } };
```

Fig. 2. Two SEDEL embeddings of circuits.

```
  (l1.above (l1.stretch (cons 2 (cons 2 nil)) (l1.fan 2))
    (l1.beside (l1.beside (l1.identity 1) (l1.fan 2)) (l1.identity 1)));
```

Here `e1` evaluates to `{width = 4}`. If we want to know the depth of the circuit, we have to replicate the code with `language2`.

Dynamically reusable circuits. Just like in Haskell, we can use polymorphism to define a type of circuits that can be interpreted with different languages.

```
type DCircuit = { accept : forall C. Circuit[C] → C };
```

In contrast to the Haskell solution, this implementation explicitly accepts the implementation.

```
brentKung : DCircuit = {
  accept C l = l.above (l.beside (l.fan 2) (l.fan 2))
    (l.above (l.stretch (cons 2 (cons 2 nil)) (l.fan 2))
      (l.beside (l.beside (l.identity 1) (l.fan 2)) (l.identity 1))) };
e1 = brentKung.accept Width language1;
e2 = brentKung.accept Depth language2;
```

Automatic composition of languages. Of course, like in Haskell we can also compute both results simultaneously. However, unlike in Haskell, the composition of the two interpretation requires no boilerplate whatsoever—in particular, there is no SEDEL counterpart of the `Circuit (c1, c2)` instance. Instead, we can just compose the two interpretations with the term-level merge operator (`,,`) and specify the desired type `Circuit[Width & Depth]`.

```
language3 : Circuit[Width & Depth] = language1 ,, language2;
e3 = brentKung.accept (Width & Depth) language3;
```

Here the use of the merge operator creates a term with the intersection type `Circuit[Width] & Circuit[Depth]`. Implicitly, the SEDEL type system takes care of the details, turning this intersection type into `Circuit[Width & Depth]`. This is possible because intersection (`&`) distributes over function and record types (a distinctive feature of BCD-style subtyping).

Composition of dependent interpretations. In SEDEL the composition scales nicely to dependent interpretations. For instance, the well-sizedness interpretation can be expressed without explicit projections.

```
type WellSized = { wS : Bool };
language4 = {
  identity (n : Int) = { wS = true },
  fan      (n : Int) = { wS = true },
  above (c1 : WellSized & Width) (c2 : WellSized & Width) =
    { wS = c1.wS && c2.wS && c1.width == c2.width },
  beside (c1 : WellSized) (c2 : WellSized) = { wS  = c1.wS && c2.wS  },
  stretch (ws : List[Int]) (c : WellSized & Width) =
    { wS = c.wS && length ws == c.width  } };
```

Here the `WellSized & Width` type in the `above` and `stretch` cases expresses that both the well-sizedness and width of subcircuits must be given, and that the width implementation is left as a dependency—when `language4` is used, then the width implementation must be provided. Again, the distributive properties of `&` in the type system take care of merging the two interpretations.

```
e4   = brentKung.accept (WellSized & Width) (language1 ,, language4);
main = e4.wS -- Output: true
```

Disjoint polymorphism and dynamic merges. While it may seem from the above examples that definitions have to be merged statically, SEDEL in fact supports dynamic merges. For instance, we can encapsulate the merge operator in the `combine` function while abstracting over the two components `x` and `y` that are merged as well as over their types `A` and `B`.

```
combine A [B * A] (x : A) (y : B) = x ,, y;
```

This way the components `x` and `y` are only known at runtime and thus the merge can only happen at that time. The types `A` and `B` cannot be chosen entirely freely. For instance, if both components would contribute an implementation for the same method, which implementation is provided by the combination would be ambiguous. To avoid this problem the two types `A` and `B` have to be *disjoint*. This is expressed in the disjointness constraint `* A` on the quantifier of the type variable `B`. If a quantifier mentions no disjointness constraint, like that of `A`, it defaults to the trivial $* \top$ constraint which implies no restriction.

3 Semantics of the F_i^+ Calculus

This section gives a formal account of F_i^+, the first typed calculus combining disjoint polymorphism [2] (and disjoint intersection types) with BCD subtyping [3].

Types	A, B, C	$::= \mathsf{Int} \mid \top \mid \bot \mid A \to B \mid A \,\&\, B \mid \{l : A\} \mid \alpha \mid \forall(\alpha * A).\, B$
Expressions	E	$::= x \mid i \mid \top \mid \lambda x.\, E \mid E_1\, E_2 \mid E_1 \,,,\, E_2 \mid E : A \mid \{l = E\} \mid E.l$
		$\mid \quad \Lambda(\alpha * A).\, E \mid E\, A$
Term contexts Γ		$::= \bullet \mid \Gamma, x : A$
Type contexts Δ		$::= \bullet \mid \Delta, \alpha * A$

Fig. 3. Syntax of F_i^+

The main differences to F_i are in the subtyping, well-formedness and disjointness relations. F_i^+ adds BCD subtyping and unrestricted intersections, and also closes an open problem of F_i by including the bottom type. The dynamic semantics of F_i^+ is given by elaboration to the target calculus F_{co}—a variant of System F extended with products and explicit coercions.

3.1 Syntax and Semantics

Figure 3 shows the syntax of F_i^+. Metavariables A, B, C range over types. Types include standard constructs from prior work [2,36]: integers Int, the top type $\top$, arrows $A \to B$, intersections $A \,\&\, B$, single-field record types $\{l : A\}$ and disjoint quantification $\forall(\alpha * A).\, B$. One novelty in F_i^+ is the addition of the uninhabited bottom type $\bot$. Metavariable E ranges over expressions. Expressions are integer literals i, the top value $\top$, lambda abstractions $\lambda x.\, E$, applications $E_1\, E_2$, merges $E_1 \,,,\, E_2$, annotated terms $E : A$, single-field records $\{l = E\}$, record projections $E.l$, type abstractions $\Lambda(\alpha * A).\, E$ and type applications $E\, A$.

Well-formedness and unrestricted intersections. F_i^+'s well-formedness judgment of types $\Delta \vdash A$ is standard, and only enforces well-scoping. This is one of the key differences from F_i, which uses well-formedness to also ensure that all intersection types are disjoint. In other words, while in F_i all valid intersection types must be disjoint, in F_i^+ unrestricted intersection types such as $\mathsf{Int} \,\&\, \mathsf{Int}$ are allowed. More specifically, the well-formedness of intersection types in F_i^+ and F_i is:

$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \,\&\, B}\ \textsc{wf-}\mathsf{F}_i^+ \qquad\qquad \frac{\Delta \vdash A \qquad \Delta \vdash B \qquad \boxed{\Delta \vdash A * B}}{\Delta \vdash A \,\&\, B}\ \textsc{wf-}\mathsf{F}_i$$

Notice that F_i has an extra disjointness condition $\Delta \vdash A * B$ in the premise. This is crucial for F_i's syntactic method for proving coherence, but also burdens the calculus with various syntactic restrictions and complicates its metatheory. For example, it requires extra effort to show that F_i only produces disjoint intersection types. As a consequence, F_i features a *weaker* substitution lemma (note the gray part in Proposition 1) than F_i^+ (Lemma 1).

Proposition 1 (Type substitution in F_i). *If $\Delta \vdash A$, $\Delta \vdash B$, $(\alpha * C) \in \Delta$, $\boxed{\Delta \vdash B * C}$ and well-formed context $[B/\alpha]\Delta$, then $[B/\alpha]\Delta \vdash [B/\alpha]A$.*

Lemma 1 (Type substitution in F_i^+). *If $\Delta \vdash A$, $\Delta \vdash B$, $(\alpha * C) \in \Delta$ and well-formed context $[B/\alpha]\Delta$, then $[B/\alpha]\Delta \vdash [B/\alpha]A$.*

$$\boxed{A <: B \rightsquigarrow co} \qquad \textit{(Declarative subtyping)}$$

$$\textsc{S-refl}\ \frac{}{A <: A \rightsquigarrow \mathsf{id}} \qquad \textsc{S-trans}\ \frac{A_2 <: A_3 \rightsquigarrow co_1 \qquad A_1 <: A_2 \rightsquigarrow co_2}{A_1 <: A_3 \rightsquigarrow co_1 \circ co_2} \qquad \textsc{S-top}\ \frac{}{A <: \top \rightsquigarrow \mathsf{top}}$$

$$\textsc{S-rcd}\ \frac{A <: B \rightsquigarrow co}{\{l : A\} <: \{l : B\} \rightsquigarrow co} \qquad \textsc{S-andl}\ \frac{}{A_1 \,\&\, A_2 <: A_1 \rightsquigarrow \pi_1} \qquad \textsc{S-andr}\ \frac{}{A_1 \,\&\, A_2 <: A_2 \rightsquigarrow \pi_2}$$

$$\textsc{S-arr}\ \frac{B_1 <: A_1 \rightsquigarrow co_1 \qquad A_2 <: B_2 \rightsquigarrow co_2}{A_1 \to A_2 <: B_1 \to B_2 \rightsquigarrow co_1 \to co_2} \qquad \textsc{S-and}\ \frac{A_1 <: A_2 \rightsquigarrow co_1 \qquad A_1 <: A_3 \rightsquigarrow co_2}{A_1 <: A_2 \,\&\, A_3 \rightsquigarrow \langle co_1, co_2 \rangle}$$

$$\textsc{S-distArr}\ \frac{}{(A_1 \to A_2) \,\&\, (A_1 \to A_3) <: A_1 \to A_2 \,\&\, A_3 \rightsquigarrow \mathsf{dist}_{\to}} \qquad \textsc{S-topArr}\ \frac{}{\top <: \top \to \top \rightsquigarrow \mathsf{top}_{\to}}$$

$$\textsc{S-distRcd}\ \frac{}{\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\} \rightsquigarrow \mathsf{id}} \qquad \textsc{S-topRcd}\ \frac{}{\top <: \{l : \top\} \rightsquigarrow \mathsf{id}} \qquad \textsc{S-bot}\ \frac{}{\bot <: A \rightsquigarrow \mathsf{bot}}$$

$$\textsc{S-forall}\ \frac{B_1 <: B_2 \rightsquigarrow co \qquad A_2 <: A_1}{\forall(\alpha * A_1).\, B_1 <: \forall(\alpha * A_2).\, B_2 \rightsquigarrow co_{\forall}} \qquad \textsc{S-topAll}\ \frac{}{\top <: \forall(\alpha * \top).\, \top \rightsquigarrow \mathsf{top}_{\forall}}$$

$$\textsc{S-distAll}\ \frac{}{(\forall(\alpha * A).\, B_1) \,\&\, (\forall(\alpha * A).\, B_2) <: \forall(\alpha * A).\, B_1 \,\&\, B_2 \rightsquigarrow \mathsf{dist}_{\forall}}$$

Fig. 4. Declarative subtyping

Declarative subtyping. F_i^+'s subtyping judgment is another major difference to F_i, because it features BCD-style subtyping and a rule for the bottom type. The full set of subtyping rules are shown in Fig. 4. The reader is advised to ignore the gray parts for now. Our subtyping rules extend the BCD-style subtyping rules from λ_i^+ [5] with a rule for parametric (disjoint) polymorphism (rule S-forall). Moreover, we have three new rules: rule S-bot for the bottom type, and rules S-distAll and S-topAll for distributivity of disjoint quantification. The subtyping relation is a partial order (rules S-refl and S-trans). Most of the rules are quite standard. $\bot$ is a subtype of all types (rule S-bot). Subtyping of disjoint quantification is covariant in its body, and contravariant in its disjointness constraints (rule S-forall). Of particular interest are those so-called "distributivity" rules: rule S-distArr says intersections distribute over arrows; rule S-distRcd says intersections distribute over records. Similarly, rule S-distAll dictates that intersections may distribute over disjoint quantifiers.

$$\boxed{\Delta;\Gamma \vdash E \Rightarrow A \rightsquigarrow e} \qquad \textit{(Inference)}$$

$$\textsc{T-top}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \langle\rangle} \qquad \textsc{T-nat}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash i \Rightarrow \mathsf{Int} \rightsquigarrow i} \qquad \textsc{T-var}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad (x : A) \in \Gamma}{\Delta;\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\textsc{T-app}\ \frac{\Delta;\Gamma \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta;\Gamma \vdash E_1\ E_2 \Rightarrow A_2 \rightsquigarrow e_1\ e_2} \qquad \textsc{T-merge}\ \frac{\Delta;\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad \Delta \vdash A_1 * A_2}{\Delta;\Gamma \vdash E_1 \,,,\, E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow \langle e_1, e_2\rangle}$$

$$\textsc{T-anno}\ \frac{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow e}{\Delta;\Gamma \vdash E : A \Rightarrow A \rightsquigarrow e} \qquad \textsc{T-rcd}\ \frac{\Delta;\Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Delta;\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow e}$$

$$\textsc{T-proj}\ \frac{\Delta;\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Delta;\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e} \qquad \textsc{T-tabs}\ \frac{\Delta, \alpha * A;\Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad \Delta \vdash A \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash \Lambda(\alpha * A).\, E \Rightarrow \forall(\alpha * A).\, B \rightsquigarrow \Lambda\alpha.\, e}$$

$$\textsc{T-tapp}\ \frac{\Delta;\Gamma \vdash E \Rightarrow \forall(\alpha * B).\, C \rightsquigarrow e \qquad \Delta \vdash A * B}{\Delta;\Gamma \vdash E\ A \Rightarrow [A/\alpha]C \rightsquigarrow e\,|A|}$$

$$\boxed{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow e} \qquad \textit{(Checking)}$$

$$\textsc{T-abs}\ \frac{\Delta \vdash A \qquad \Delta;\Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Delta;\Gamma \vdash \lambda x.\, E \Leftarrow A \rightarrow B \rightsquigarrow \lambda x.\, e} \qquad \textsc{T-sub}\ \frac{\Delta;\Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad B <: A \rightsquigarrow co}{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow co\ e}$$

Fig. 5. Bidirectional type system

Typing rules. F_i^+ features a bidirectional type system inherited from F_i. The full set of typing rules are shown in Fig. 5. Again we ignore the gray parts and explain them in Sect. 3.3. The inference judgment $\Delta;\Gamma \vdash E \Rightarrow A$ says that we can synthesize the type A under the contexts Δ and Γ. The checking judgment $\Delta;\Gamma \vdash E \Leftarrow A$ asserts that E checks against the type A under the contexts Δ and Γ. Most of the rules are quite standard in the literature. The merge expression $E_1 \,,,\, E_2$ is well-typed if both sub-expressions are well-typed, and their types are *disjoint* (rule T-MERGE). The disjointness relation will be explained in Sect. 3.2. To infer a type abstraction (rule T-TABS), we add disjointness constraints to the type context. For a type application (rule T-TAPP), we check that the type argument satisfies the disjointness constraints. Rules T-MERGE and T-TAPP are the only rules checking disjointness.

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

$$\textsc{TL-top}\ \frac{}{\rceil \top \lceil} \qquad \textsc{TL-and}\ \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil} \qquad \textsc{TL-arr}\ \frac{\rceil B \lceil}{\rceil A \to B \lceil} \qquad \textsc{TL-rcd}\ \frac{\rceil A \lceil}{\rceil \{l : A\} \lceil} \qquad \textsc{TL-all}\ \frac{\rceil B \lceil}{\rceil \forall (\alpha * A).\, B \lceil}$$

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

$$\textsc{D-topL}\ \frac{\rceil A \lceil}{\Delta \vdash A * B} \qquad \textsc{D-topR}\ \frac{\rceil B \lceil}{\Delta \vdash A * B} \qquad \textsc{D-arr}\ \frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2}$$

$$\textsc{D-andL}\ \frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B} \qquad \textsc{D-andR}\ \frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2}$$

$$\textsc{D-rcdEq}\ \frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}} \qquad \textsc{D-rcdNeq}\ \frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}} \qquad \textsc{D-tvarL}\ \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B}$$

$$\textsc{D-tvarR}\ \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha} \qquad \textsc{D-forall}\ \frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall (\alpha * A_1).\, B_1 * \forall (\alpha * A_2).\, B_2} \qquad \textsc{D-ax}\ \frac{A *_{ax} B}{\Delta \vdash A * B}$$

Fig. 6. Selected rules for disjointness

3.2 Disjointness

We now turn to another core judgment of F_i^+—the disjointness relation, shown in Fig. 6. The disjointness rules are mostly inherited from F_i [2], but the new bottom type requires a notable change regarding disjointness with *top-like types.*

Top-like types. Top-like types are all types that are isomorphic to $\top$ (i.e., simultaneously sub- and supertypes of $\top$). Hence, they are inhabited by a single value, isomorphic to the $\top$ value. Figure 6 captures this notion in a syntax-directed fashion in the $\rceil A \lceil$ predicate. As a historical note, the concept of top-like types was already known by Barendregt et al. [3]. The λ_i calculus [36] re-discovered it and coined the term "top-like types"; the F_i calculus [2] extended it with universal quantifiers. Note that in both calculi, top-like types are solely employed for enabling a syntactic method of proving coherence, and due to the lack of BCD subtyping, they do not have a type-theoretic interpretation of top-like types.

Disjointness rules. The disjointness judgment $\Delta \vdash A * B$ is helpful to check whether the merge of two expressions of type A and B preserves coherence. Incoherence arises when both expressions produce distinct values for the same type, either directly when they are both of that same type, or through implicit

Types	τ	$::= \mathsf{Int} \mid \langle\rangle \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \alpha \mid \forall\alpha.\,\tau$
Terms	e	$::= x \mid i \mid \langle\rangle \mid \lambda x.\,e \mid e_1\;e_2 \mid \langle e_1, e_2\rangle \mid \Lambda\alpha.\,e \mid e\;\tau \mid co\;e$
Coercions	co	$::= \mathsf{id} \mid co_1 \circ co_2 \mid \mathsf{top} \mid \mathsf{bot} \mid co_1 \to co_2 \mid \langle co_1, co_2\rangle \mid \pi_1 \mid \pi_2$
		$\mid\;\; co_\forall \mid \mathsf{dist}_\to \mid \mathsf{top}_\to \mid \mathsf{top}_\forall \mid \mathsf{dist}_\forall$
Values	v	$::= i \mid \langle\rangle \mid \lambda x.\,e \mid \langle v_1, v_2\rangle \mid \Lambda\alpha.\,e \mid (co_1 \to co_2)\;v \mid co_\forall\;v$
		$\mid\;\; \mathsf{dist}_\to\;v \mid \mathsf{top}_\to\;v \mid \mathsf{top}_\forall\;v \mid \mathsf{dist}_\forall\;v$
Term contexts	Ψ	$::= \bullet \mid \Psi, x : \tau$
Type contexts	Φ	$::= \bullet \mid \Phi, \alpha$
Evaluation contexts	$\mathcal{E}$	$::= [\cdot] \mid \mathcal{E}\;e \mid v\;\mathcal{E} \mid \langle\mathcal{E}, e\rangle \mid \langle v, \mathcal{E}\rangle \mid co\,\mathcal{E} \mid \mathcal{E}\;\tau$

Fig. 7. Syntax of F_{co}

upcasting to a common supertype. Of course we can safely disregard top-like types in this matter because they do not have two distinct values. In short, it suffices to check that the two types have only top-like supertypes in common.

Because $\bot$ and any another type A always have A as a common supertype, it follows that $\bot$ is only disjoint to A when A is top-like. More generally, if A is a top-like type, then A is disjoint to any type. This is the rationale behind the two rules D-topL and D-topR, which generalize and subsume $\Delta \vdash \top * A$ and $\Delta \vdash A * \top$ from F_i, and also cater to the bottom type. Two other interesting rules are D-tvarL and D-tvarR, which dictate that a type variable α is disjoint with some type B if its disjointness constraints A is a subtype of B. Disjointness axioms $A *_{ax} B$ (appearing in rule D-ax) take care of two types with different type constructors (e.g., Int and records). Axiom rules can be found in the appendix. Finally we note that the disjointness relation is symmetric.

3.3 Elaboration and Type Safety

The dynamic semantics of F_i^+ is given by elaboration into a target calculus. The target calculus F_{co} is the standard call-by-value System F extended with products and coercions. The syntax of F_{co} is shown in Fig. 7.

Type translation. Definition 1 defines the type translation function $|\cdot|$ from F_i^+ types A to F_{co} types τ. Most cases are straightforward. For example, $\bot$ is mapped to an uninhabited type $\forall\alpha.\,\alpha$; disjoint quantification is mapped to universal quantification, dropping the disjointness constraints. $|\cdot|$ is naturally extended to work on contexts as well.

Definition 1. *Type translation $|\cdot|$ is defined as follows:*

$$|\mathsf{Int}| = \mathsf{Int} \qquad |\top| = \langle\rangle \qquad |A \to B| = |A| \to |B|$$
$$|A \,\&\, B| = |A| \times |B| \qquad |\{l : A\}| = |A| \qquad |\alpha| = \alpha$$
$$|\bot| = \forall\alpha.\,\alpha \qquad |\forall(\alpha * A).\,B| = \forall\alpha.\,|B|$$

$\boxed{e \longrightarrow e'}$ *(Single-step reduction)*

$$\textsc{r-forall}\ \frac{}{(co_{\forall}\ v)\ \tau \longrightarrow co\ (v\ \tau)} \qquad \textsc{r-topAll}\ \frac{}{(\mathsf{top}_{\forall}\ \langle\rangle)\ \tau \longrightarrow \langle\rangle} \qquad \textsc{r-distAll}\ \frac{}{(\mathsf{dist}_{\forall}\ \langle v_1, v_2\rangle)\ \tau \longrightarrow \langle v_1\ \tau, v_2\ \tau\rangle}$$

$$\textsc{r-tapp}\ \frac{}{(\Lambda\alpha.\ e)\ \tau \longrightarrow [\tau/\alpha]e} \qquad \textsc{r-app}\ \frac{}{(\lambda x.\ e)\ v \longrightarrow [v/x]e} \qquad \textsc{r-ctxt}\ \frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

Fig. 8. Selected reduction rules

Coercions and coercive subtyping. We follow prior work [5,6] by having a syntactic category for coercions [22]. In Fig. 7, we have several new coercions: bot, $co_{\forall}$, $\mathsf{dist}_{\forall}$ and $\mathsf{top}_{\forall}$ due to the addition of polymorphism and bottom type. As seen in Fig. 4 the coercive subtyping judgment has the form $A <: B \rightsquigarrow co$, which says that the subtyping derivation for $A <: B$ produces a coercion co that converts terms of type $|A|$ to $|B|$.

F_{co} *static semantics.* The typing rules of F_{co} are quite standard. We have one rule T-CAPP regarding coercion application, which uses the judgment $co{::}\tau \rhd \tau'$ to type coercions. We show two representative rules CT-FORALL and CT-BOT.

$$\textsc{t-capp}\ \frac{\Phi; \Psi \vdash e : \tau \qquad co :: \tau \rhd \tau'}{\Phi; \Psi \vdash co\ e : \tau'} \qquad \textsc{ct-forall}\ \frac{co :: \tau_1 \rhd \tau_2}{co_{\forall} :: \forall\alpha.\ \tau_1 \rhd \forall\alpha.\ \tau_2} \qquad \textsc{ct-bot}\ \frac{}{\mathsf{bot} :: \forall\alpha.\ \alpha \rhd \tau}$$

F_{co} *dynamic semantics.* The dynamic semantics of F_{co} is mostly unremarkable. We write $e \longrightarrow e'$ to mean one-step reduction. Figure 8 shows selected reduction rules. The first line shows three representative rules regarding coercion reductions. They do not contribute to computation but merely rearrange coercions. Our coercion reduction rules are quite standard but not efficient in terms of space. Nevertheless, there is existing work on space-efficient coercions [23,50], which should be applicable to our work as well. Rule R-APP is the usual β-rule that performs actual computation, and rule R-CTXT handles reduction under an evaluation context. As usual, $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$. Now we can show that F_{co} is type safe:

Theorem 1 (Preservation). *If* $\bullet; \bullet \vdash e : \tau$ *and* $e \longrightarrow e'$*, then* $\bullet; \bullet \vdash e' : \tau$*.*

Theorem 2 (Progress). *If* $\bullet; \bullet \vdash e : \tau$*, either* e *is a value, or* $\exists e'.\ e \longrightarrow e'$*.*

Elaboration. Now consider the translation parts in Fig. 5. The key idea of the translation follows the prior work [2,5,16,36]: merges are elaborated to pairs (rule T-MERGE); disjoint quantification and disjoint type applications (rules T-TABS and T-TAPP)) are elaborated to regular universal quantification and type applications, respectively. Finally, the following lemma connects F_i^+ to F_{co}:

Lemma 2 (Elaboration soundness). *We have that:*

- *If* $A <: B \rightsquigarrow co$, *then* $co :: |A| \triangleright |B|$.
- *If* $\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e$, *then* $|\Delta|; |\Gamma| \vdash e : |A|$.
- *If* $\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e$, *then* $|\Delta|; |\Gamma| \vdash e : |A|$.

4 Algorithmic System and Decidability

The subtyping relation in Fig. 4 is highly non-algorithmic due to the presence of a transitivity rule. This section presents an alternative algorithmic formulation. Our algorithm extends that of λ_i^+, which itself was inspired by Pierce's decision procedure [38], to handle disjoint quantifiers and the bottom type. We then prove that the algorithm is sound and complete with respect to declarative subtyping.

Additionally we prove that the subtyping and disjointness relations are decidable. Although the proofs of this fact are fairly straightforward, it is nonetheless remarkable since it contrasts with the subtyping relation for (full) $\mathsf{F}_{<:}$ [10], which is undecidable [40]. Thus while bounded quantification is infamous for its undecidability, disjoint quantification has the nicer property of being decidable.

4.1 Algorithmic Subtyping Rules

While Fig. 4 is a fine specification of how subtyping should behave, it cannot be read directly as a subtyping algorithm for two reasons: (1) the conclusions of rules S-REFL and S-TRANS overlap with the other rules, and (2) the premises of rule S-TRANS mention a type that does not appear in the conclusion. Simply dropping the two offending rules from the system is not possible without losing expressivity [29]. Thus we need a different approach. Following λ_i^+, we intend the algorithmic judgment $\mathcal{Q} \vdash A <: B$ to be equivalent to $A <: \mathcal{Q} \Rightarrow B$, where $\mathcal{Q}$ is a queue used to track record labels, domain types and disjointness constraints. The full rules of the algorithmic subtyping of F_i^+ are shown Fig. 9.

Definition 2 ($\mathcal{Q} ::= [] \mid l, \mathcal{Q} \mid B, \mathcal{Q} \mid \alpha * B, \mathcal{Q}$)**.** $\mathcal{Q} \Rightarrow A$ *is defined as follows:*

$[] \Rightarrow A = A$	$(B, \mathcal{Q}) \Rightarrow A = B \to (\mathcal{Q} \Rightarrow A)$
$(l, \mathcal{Q}) \Rightarrow A = \{l : \mathcal{Q} \Rightarrow A\}$	$(\alpha * B, \mathcal{Q}) \Rightarrow A = \forall(\alpha * B).\, \mathcal{Q} \Rightarrow A$

For brevity of the algorithm, we use metavariable c to mean type constants:

$$\text{Type Constants} \quad c ::= \mathsf{Int} \mid \bot \mid \alpha$$

The basic idea of $\mathcal{Q} \vdash A <: B$ is to perform a case analysis on B until it reaches type constants. We explain new rules regarding disjoint quantification and the bottom type. When a quantifier is encountered in B, rule A-FORALL pushes the type variables with its disjointness constraints onto $\mathcal{Q}$ and continue with the body. Correspondingly, in rule A-ALLCONST, when a quantifier is encountered in A, and the head of $\mathcal{Q}$ is a type variable, this variable is popped out and we continue with the body. Rule A-BOT is similar to its declarative counterpart. Two meta-functions $[\![\mathcal{Q}]\!]^{\top}$ and $[\![\mathcal{Q}]\!]^{\&}$ are meant to generate correct forms of coercions, and their definitions are shown in the appendix. For other algorithmic rules, we refer to λ_i^+ [5] for detailed explanations.

$\boxed{\mathcal{Q} \vdash A <: B \rightsquigarrow co}$ *(Algorithmic subtyping)*

$$\textsc{A-top}\quad \frac{}{\mathcal{Q} \vdash A <: \top \rightsquigarrow [\![\mathcal{Q}]\!]^{\top} \circ \mathsf{top}}$$

$$\textsc{A-and}\quad \frac{\mathcal{Q} \vdash A <: B_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A <: B_2 \rightsquigarrow co_2}{\mathcal{Q} \vdash A <: B_1 \,\&\, B_2 \rightsquigarrow [\![\mathcal{Q}]\!]^{\&} \circ \langle co_1, co_2 \rangle}$$

$$\textsc{A-arr}\quad \frac{\mathcal{Q}, B_1 \vdash A <: B_2 \rightsquigarrow co}{\mathcal{Q} \vdash A <: B_1 \to B_2 \rightsquigarrow co}$$

$$\textsc{A-rcd}\quad \frac{\mathcal{Q}, l \vdash A <: B \rightsquigarrow co}{\mathcal{Q} \vdash A <: \{l : B\} \rightsquigarrow co}$$

$$\textsc{A-forall}\quad \frac{\mathcal{Q}, \alpha * B_1 \vdash A <: B_2 \rightsquigarrow co}{\mathcal{Q} \vdash A <: \forall(\alpha * B_1).\, B_2 \rightsquigarrow co}$$

$$\textsc{A-const}\quad \frac{}{[] \vdash c <: c \rightsquigarrow \mathsf{id}}$$

$$\textsc{A-bot}\quad \frac{}{\mathcal{Q} \vdash \bot <: c \rightsquigarrow \mathsf{bot}}$$

$$\textsc{A-arrConst}\quad \frac{[] \vdash A <: A_1 \rightsquigarrow co_1 \qquad \mathcal{Q} \vdash A_2 <: c \rightsquigarrow co_2}{A, \mathcal{Q} \vdash A_1 \to A_2 <: c \rightsquigarrow co_1 \to co_2}$$

$$\textsc{A-rcdConst}\quad \frac{\mathcal{Q} \vdash A <: c \rightsquigarrow co}{l, \mathcal{Q} \vdash \{l : A\} <: c \rightsquigarrow co}$$

$$\textsc{A-andConst}\quad \frac{\mathcal{Q} \vdash A_i <: c \rightsquigarrow co \qquad i \in \{1, 2\}}{\mathcal{Q} \vdash A_1 \,\&\, A_2 <: c \rightsquigarrow co \circ \pi_i}$$

$$\textsc{A-allConst}\quad \frac{[] \vdash A <: A_1 \qquad \mathcal{Q} \vdash A_2 <: c \rightsquigarrow co}{(\alpha * A, \mathcal{Q}) \vdash \forall(\alpha * A_1).\, A_2 <: c \rightsquigarrow co_{\forall}}$$

Fig. 9. Algorithmic subtyping

Correctness of the algorithm. We prove that the algorithm is sound and complete with respect to the specification. We refer the reader to our Coq formalization for more details. We only show the two major theorems:

Theorem 3 (Soundness). *If $\mathcal{Q} \vdash A <: B \rightsquigarrow co$ then $A <: \mathcal{Q} \Rightarrow B \rightsquigarrow co$.*

Theorem 4 (Completeness). *If $A <: B \rightsquigarrow co$, then $\exists co'.\ [] \vdash A <: B \rightsquigarrow co'$.*

4.2 Decidability

Moreover, we prove that our algorithmic type system is decidable. To see this, first notice that the bidirectional type system is syntax-directed, so we only need to show decidability of algorithmic subtyping and disjointness. The full (manual) proofs for decidability can be found in the appendix.

Lemma 3 (Decidability of algorithmic subtyping). *Given $\mathcal{Q}$, A and B, it is decidable whether there exists co, such that $\mathcal{Q} \vdash A <: B \rightsquigarrow co$.*

Lemma 4 (Decidability of disjointness checking). *Given Δ, A and B, it is decidable whether $\Delta \vdash A * B$.*

One interesting observation here is that although our disjointness quantification has a similar shape to bounded quantification $\forall(\alpha <: A).\, B$ in $\mathsf{F}_{<:}$ [10], subtyping for $\mathsf{F}_{<:}$ is undecidable [40]. In $\mathsf{F}_{<:}$, the subtyping relation between bounded quantification is:

$$\frac{\Delta \vdash A_2 <: A_1 \qquad \Delta, \alpha <: A_2 \vdash B_1 <: B_2}{\Delta \vdash \forall(\alpha <: A_1).\, B_1 <: \forall(\alpha <: A_2).\, B_2}\ \text{FSUB-FORALL}$$

Compared with rule S-FORALL, both rules are contravariant on bounded/disjoint types, and covariant on the body. However, with bounded quantification it is fundamental to track the bounds in the environment, which complicates the design of the rules and makes subtyping undecidable with rule FSUB-FORALL. Decidability can be recovered by employing an invariant rule for bounded quantification (that is by forcing A_1 and A_2 to be identical). Disjoint quantification does not require such invariant rule for decidability.

5 Establishing Coherence for F_i^+

In this section, we establish the coherence property for F_i^+. The proof strategy mostly follows that of λ_i^+, but the construction of the heterogeneous logical relation is significantly more complicated. Firstly in Sect. 5.1 we discuss why adding BCD subtyping to disjoint polymorphism introduces significant complications. In Sect. 5.2, we discuss why a natural extension of System F's logical relation to deal with disjoint polymorphism fails. The technical difficulty is *well-foundedness*, stemming from the interaction between impredicativity and disjointness. Finally in Sect. 5.3, we present our (predicative) logical relation that is specially crafted to prove coherence for F_i^+.

5.1 The Challenge

Before we tackle the coherence of F_i^+, let us first consider how F_i (and its predecessor λ_i) enforces coherence. Its essentially syntactic approach is to make sure that there is at most one subtyping derivation for any two types. As an immediate consequence, the produced coercions are uniquely determined and thus the calculus is clearly coherent. Key to this approach is the invariant that the type system only produces *disjoint* intersection types. As we mentioned in Sect. 3, this invariant complicates the calculus and its metatheory, and leads to a weaker substitution lemma. Moreover, the syntactic coherence approach is incompatible with BCD subtyping, which leads to multiple subtyping derivations with different coercions and requires a more general substitution lemma. To accommodate BCD into λ_i, Bi et al. [5] have created the λ_i^+ calculus and developed a semantically-founded proof method based on logical relations. Because λ_i^+ does not feature polymorphism, the problem at hand is to incorporate support for polymorphism in this semantic approach to coherence, which turns out to be more challenging than is apparent.

$$\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}; \mathsf{Int}]\!] &\triangleq \exists i.\, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\tau_1 \to \tau_2; \tau_1' \to \tau_2']\!] &\triangleq \forall (v, v') \in \mathcal{V}[\![\tau_1; \tau_1']\!].\, (v_1\, v, v_2\, v') \in \mathcal{E}[\![\tau_2; \tau_2']\!] \\
(\langle v_1, v_2\rangle, v_3) \in \mathcal{V}[\![\tau_1 \times \tau_2; \tau_3]\!] &\triangleq (v_1, v_3) \in \mathcal{V}[\![\tau_1; \tau_3]\!] \wedge (v_2, v_3) \in \mathcal{V}[\![\tau_2; \tau_3]\!] \\
(v_3, \langle v_1, v_2\rangle) \in \mathcal{V}[\![\tau_3; \tau_1 \times \tau_2]\!] &\triangleq (v_3, v_1) \in \mathcal{V}[\![\tau_3; \tau_1]\!] \wedge (v_3, v_2) \in \mathcal{V}[\![\tau_3; \tau_2]\!]
\end{aligned}$$

Fig. 10. Selected cases from λ_i^+'s canonicity relation

5.2 Impredicativity and Disjointness at Odds

Figure 10 shows selected cases of *canonicity*, which is λ_i^+'s (heterogeneous) logical relation used in the coherence proof. The definition captures that two values v_1 and v_2 of types τ_1 and τ_2 are in $\mathcal{V}[\![\tau_1; \tau_2]\!]$ iff either the types are disjoint or the types are equal and the values are semantically equivalent. Because both alternatives entail coherence, canonicity is key to λ_i^+'s coherence proof.

Well-foundedness issues. For F_i^+, we need to extend canonicity with additional cases to account for universally quantified types. For reasons that will become clear in Sect. 5.3, the type indices become source types (rather than target types as in Fig. 10). A naive formulation of one case rule is:

$$\begin{aligned}
&(v_1, v_2) \in \mathcal{V}[\![\forall(\alpha * A_1).\, B_1; \forall(\alpha * A_2).\, B_2]\!] \triangleq \\
&\qquad \forall C_1 * A_1, C_2 * A_2.\ (v_1\, |C_1|, v_2\, |C_2|) \in \mathcal{E}[\![[C_1/\alpha]B_1; [C_2/\alpha]B_2]\!]
\end{aligned}$$

This case is problematic because it destroys the well-foundedness of λ_i^+'s logical relation, which is based on structural induction on the type indices. Indeed, the type $[C_1/\alpha]B_1$ may well be larger than $\forall(\alpha * A_1).\, B_1$.

However, System F's well-known parametricity logical relation [43] provides us with a means to avoid this problem. Rather than performing the type substitution immediately as in the above rule, we can defer it to a later point by adding it to an extra parameter ρ of the relation, which accumulates the deferred substitutions. This yields a modified rule where the type indices in the recursive occurrences are indeed smaller:

$$\begin{aligned}
&(v_1, v_2) \in \mathcal{V}[\![\forall(\alpha * A_1).\, B_1; \forall(\alpha * A_2).\, B_2]\!]_\rho \triangleq \\
&\qquad \forall C_1 * A_1, C_2 * A_2. (v_1\, |C_1|, v_2\, |C_2|) \in \mathcal{E}[\![B_1; B_2]\!]_{\rho[\alpha \mapsto (C_1, C_2)]}
\end{aligned}$$

Of course, the deferred substitution has to be performed eventually, to be precise when the type indices are type variables.

$$(v_1, v_2) \in \mathcal{V}[\![\alpha; \alpha]\!]_\rho \triangleq (v_1, v_2) \in \mathcal{V}[\![\rho_1(\alpha); \rho_2(\alpha)]\!]_\emptyset$$

Unfortunately, this way we have not only moved the type substitution to the type variable case, but also the ill-foundedness problem. Indeed, this problem is also present in System F. The standard solution is to not fix the relation R by which values at type α are related to $\mathcal{V}[\![\rho_1(\alpha); \rho_2(\alpha)]\!]$, but instead to make it a parameter that is tracked by ρ. This yields the following two rules for disjoint quantification and type variables:

$$(v_1, v_2) \in \mathcal{V}[\![\forall(\alpha * A_1).\, B_1; \forall(\alpha * A_2).\, B_2]\!]_\rho \triangleq \forall C_1 * A_1, C_2 * A_2, \mathsf{R} \subseteq C_1 \times C_2.$$
$$(v_1\,|C_1|, v_2\,|C_2|) \in \mathcal{E}[\![B_1; B_2]\!]_{\rho[\alpha\mapsto(C_1,C_2,\mathsf{R})]}$$
$$(v_1, v_2) \in \mathcal{V}[\![\alpha; \alpha]\!]_\rho \triangleq (v_1, v_2) \in \rho_{\mathsf{R}}(\alpha)$$

Now we have finally recovered the well-foundedness of the relation. It is again structurally inductive on the size of the type indexes.

Heterogeneous issues. We have not yet accounted for one major difference between the parametricity relation, from which we have borrowed ideas, and the canonicity relation, to which we have been adding. The former is homogeneous (i.e., the types of the two values is the same) and therefore has one type index, while the latter is heterogeneous (i.e., the two values may have different types) and therefore has two type indices. Thus we must also consider cases like $\mathcal{V}[\![\alpha; \mathsf{Int}]\!]$. A definition that seems to handle this case appropriately is:

$$(v_1, v_2) \in \mathcal{V}[\![\alpha; \mathsf{Int}]\!]_\rho \triangleq (v_1, v_2) \in \mathcal{V}[\![\rho_1(\alpha); \mathsf{Int}]\!]_\emptyset \tag{1}$$

Here is an example to motivate it. Let $E = \Lambda(\alpha*\top).\,((\lambda x.\, x) : \alpha \,\&\, \mathsf{Int} \to \alpha \,\&\, \mathsf{Int})$. We expect that $E\,\mathsf{Int}\,1$ evaluates to $\langle 1, 1\rangle$. To prove that, we need to show $(1, 1) \in \mathcal{V}[\![\alpha; \mathsf{Int}]\!]_{[\alpha\mapsto(\mathsf{Int},\mathsf{Int},\mathsf{R})]}$. According to Eq. (1), this is indeed the case. However, we run into ill-foundedness issue again, because $\rho_1(\alpha)$ could be larger than α. Alas, this time the parametricity relation has no solution for us.

5.3 The Canonicity Relation for F_i^+

In light of the fact that substitution in the logical relation seems unavoidable in our setting, and that impredicativity is at odds with substitution, we turn to *predicativity*: we change rule T-TAPP to its predicative version:

$$\frac{\Delta; \Gamma \vdash E \Rightarrow \forall(\alpha * B).\, C \rightsquigarrow e \qquad \Delta \vdash t * B}{\Delta; \Gamma \vdash E\, t \Rightarrow [t/\alpha]C \rightsquigarrow e\,|t|}\ \textsc{T-tappMono}$$

where metavariable t ranges over monotypes (types minus disjoint quantification). We do not believe that predicativity is a severe restriction in practice, since many source languages (e.g., those based on the Hindley-Milner type system [24,32] like Haskell and OCaml) are themselves predicative and do not require the full generality of an impredicative core language.

Luckily, substitution with monotypes does not prevent well-foundedness. Figure 11 defines the *canonicity* relation for F_i^+. The canonicity relation is a family of binary relations over F_{co} values that are *heterogeneous*, i.e., indexed by two F_i^+ types. Two points are worth mentioning. (1) An apparent difference from λ_i^+'s logical relation is that our relation is now indexed by *source types*. The

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}[\![\mathsf{Int}; \mathsf{Int}]\!] &\triangleq \exists i.\, v_1 = v_2 = i \\
(v_1, v_2) \in \mathcal{V}[\![\{l : A\}; \{l : B\}]\!] &\triangleq (v_1, v_2) \in \mathcal{V}[\![A; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A_1 \to B_1; A_2 \to B_2]\!] &\triangleq \forall (v_2', v_1') \in \mathcal{V}[\![A_2; A_1]\!].\, (v_1\, v_1', v_2\, v_2') \in \mathcal{E}[\![B_1; B_2]\!] \\
(\langle v_1, v_2 \rangle, v_3) \in \mathcal{V}[\![A \,\&\, B; C]\!] &\triangleq (v_1, v_3) \in \mathcal{V}[\![A; C]\!] \wedge (v_2, v_3) \in \mathcal{V}[\![B; C]\!] \\
(v_3, \langle v_1, v_2 \rangle) \in \mathcal{V}[\![C; A \,\&\, B]\!] &\triangleq (v_3, v_1) \in \mathcal{V}[\![C; A]\!] \wedge (v_3, v_2) \in \mathcal{V}[\![C; B]\!] \\
(v_1, v_2) \in \mathcal{V}[\![\forall (\alpha * A_1).\, B_1; \forall (\alpha * A_2).\, B_2]\!] &\triangleq \forall \bullet \vdash t * A_1 \,\&\, A_2.\; (v_1\, |t|, v_2\, |t|) \in \mathcal{E}[\![[t/\alpha] B_1; [t/\alpha] B_2]\!] \\
(v_1, v_2) \in \mathcal{V}[\![A; B]\!] &\triangleq \mathsf{true} \quad \text{otherwise} \\
(e_1, e_2) \in \mathcal{E}[\![A; B]\!] &\triangleq \exists v_1, v_2.\, e_1 \longrightarrow^* v_1 \wedge e_2 \longrightarrow^* v_2 \;\wedge (v_1, v_2) \in \mathcal{V}[\![A; B]\!]
\end{aligned}
$$

$$
\rho \in \mathcal{D}[\![\Delta]\!] \triangleq \frac{}{\emptyset \in \mathcal{D}[\![\bullet]\!]} \qquad \frac{\rho \in \mathcal{D}[\![\Delta]\!] \qquad \bullet \vdash t * \rho(B)}{\rho[\alpha \mapsto t] \in \mathcal{D}[\![\Delta, \alpha * B]\!]}
$$

$$
(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\rho \triangleq \frac{}{(\emptyset, \emptyset) \in \mathcal{G}[\![\bullet]\!]_\rho} \qquad \frac{(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\rho \qquad (v_1, v_2) \in \mathcal{V}[\![\rho(A); \rho(A)]\!]}{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \in \mathcal{G}[\![\Gamma, x : A]\!]_\rho}
$$

Fig. 11. The canonicity relation for F_i^+

reason is that the type translation function (Definition 1) discards disjointness constraints, which are crucial in our setting, whereas λ_i^+'s type translation does not have information loss. (2) Heterogeneity allows relating values of different types, and in particular values whose types are disjoint. The rationale behind the canonicity relation is to combine equality checking from traditional (homogeneous) logical relations with disjointness checking. It consists of two relations: the value relation $\mathcal{V}[\![A; B]\!]$ relates *closed* values; and the expression relation $\mathcal{E}[\![A; B]\!]$—defined in terms of the value relation—relates closed expressions.

The relation $\mathcal{V}[\![A; B]\!]$ is defined by induction on the structures of A and B. For integers, it requires the two values to be literally the same. For two records to behave the same, their fields must behave the same. For two functions to behave the same, they are required to produce outputs related at B_1 and B_2 when given related inputs at A_1 and A_2. For the next two cases regarding intersection types, the relation distributes over intersection constructor $\&$. Of particular interest is the case for disjoint quantification. Notice that it *does not* quantify over arbitrary relations, but directly substitutes α with monotype t in B_1 and B_2. This means that our canonicity relation *does not* entail parametricity. However, it suffices for our purposes to prove coherence. Another noticeable thing is that we keep the invariant that A and B are closed types throughout the relation, so we no longer need to consider type variables. This simplifies things a lot. Note that when one type is $\bot$, two values are vacuously related because there simply are no values of type $\bot$. We need to show that the relation is indeed well-founded:

Lemma 5 (Well-foundedness). *The canonicity relation of* F_i^+ *is well-founded.*

Proof. Let $|\cdot|_\forall$ and $|\cdot|_s$ be the number of $\forall$-quantifies and the size of types, respectively. Consider the measure $\langle |\cdot|_\forall, |\cdot|_s \rangle$, where $\langle \dots \rangle$ denotes lexicographic order. For the case of disjoint quantification, the number of $\forall$-quantifiers decreases. For the other cases, the measure of $|\cdot|_\forall$ does not increase, and the measure of $|\cdot|_s$ strictly decreases. $\square$

5.4 Establishing Coherence

Logical equivalence. The canonicity relation can be lifted to open expressions in the standard way, i.e., by considering all possible interpretations of free type and term variables. The logical interpretations of type and term contexts are found in the bottom half of Fig. 11.

Definition 3 (Logical equivalence $\backsimeq_{log}$)

$$\begin{aligned}&\Delta;\Gamma \vdash e_1 \backsimeq_{log} e_2 : A; B \triangleq |\Delta|;|\Gamma| \vdash e_1 : |A| \wedge |\Delta|;|\Gamma| \vdash e_2 : |B| \wedge \\ &\quad (\forall \rho, \gamma_1, \gamma_2.\ \rho \in \mathcal{D}[\![\Delta]\!] \wedge (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]_\rho \Longrightarrow (\gamma_1(\rho_1(e_1)), \gamma_2(\rho_2(e_2))) \in \mathcal{E}[\![\rho(A); \rho(B)]\!])\end{aligned}$$

For conciseness, we write $\Delta;\Gamma \vdash e_1 \backsimeq_{log} e_2 : A$ to mean $\Delta;\Gamma \vdash e_1 \backsimeq_{log} e_2 : A; A$.

Contextual equivalence. Following λ_i^+, the notion of coherence is based on *contextual equivalence.* The intuition is that two programs are equivalent if we *cannot* tell them apart in any context. As usual, contextual equivalence is expressed using *expression contexts* ($\mathcal{C}$ and $\mathcal{D}$ denote F_i^+ and F_{co} expression contexts, respectively), Due to the bidirectional nature of the type system, the typing judgment of $\mathcal{C}$ features 4 different forms (full rules are in the appendix), e.g., $\mathcal{C} : (\Delta;\Gamma \Rightarrow A) \mapsto (\Delta';\Gamma' \Rightarrow A') \rightsquigarrow \mathcal{D}$ reads if $\Delta;\Gamma \vdash E \Rightarrow A$ then $\Delta';\Gamma' \vdash \mathcal{C}\{E\} \Rightarrow A'$. The judgment also generates a well-typed F_{co} context $\mathcal{D}$. The following two definitions capture the notion of contextual equivalence:

Definition 4 (Kleene Equality $\backsimeq$). *Two complete programs (i.e., closed terms of type* Int*), e and e', are Kleene equal, written $e \backsimeq e'$, iff there exists an integer i such that $e \longrightarrow^* i$ and $e' \longrightarrow^* i$.*

Definition 5 (Contextual Equivalence $\backsimeq_{ctx}$)

$$\begin{aligned}&\Delta;\Gamma \vdash E_1 \backsimeq_{ctx} E_2 : A \triangleq \forall e_1, e_2.\ \Delta;\Gamma \vdash E_1 \Rightarrow A \rightsquigarrow e_1 \wedge \Delta;\Gamma \vdash E_2 \Rightarrow A \rightsquigarrow e_2 \wedge \\ &\qquad (\forall C, \mathcal{D}.\ \mathcal{C} : (\Delta;\Gamma \Rightarrow A) \mapsto (\bullet;\bullet \Rightarrow \mathsf{Int}) \rightsquigarrow \mathcal{D} \Longrightarrow \mathcal{D}\{e_1\} \backsimeq \mathcal{D}\{e_2\})\end{aligned}$$

Coherence. For space reasons, we directly show the coherence statement of F_i^+. We need several technical lemmas such as compatibility lemmas, fundamental property, etc. The interested reader can refer to our Coq formalization.

Theorem 5 (Coherence). *We have that*

– *If $\Delta;\Gamma \vdash E \Rightarrow A$ then $\Delta;\Gamma \vdash E \backsimeq_{ctx} E : A$.*
– *If $\Delta;\Gamma \vdash E \Leftarrow A$ then $\Delta;\Gamma \vdash E \backsimeq_{ctx} E : A$.*

That is, coherence is a special case of Definition 5 where E_1 and E_2 are the same. At first glance, this appears underwhelming: of course E behaves the same as itself! The tricky part is that, if we expand it according to Definition 5, it is not E itself but all its translations e_1 and e_2 that behave the same!

6 Related Work

Coherence. In calculi featuring coercive subtyping, a semantics that interprets the subtyping judgment by introducing explicit coercions is typically defined on typing derivations rather than on typing judgments. A natural question that arises for such systems is whether the semantics is *coherent*, i.e., distinct typing derivations of the same typing judgment possess the same meaning. Since Reynolds [45] proved the coherence of a calculus with intersection types, many researchers have studied the problem of coherence in a variety of typed calculi. Two approaches are commonly found in the literature. The first approach is to find a normal form for a representation of the derivation and show that normal forms are unique for a given typing judgment [8,15,47]. However, this approach cannot be directly applied to Curry-style calculi (where the lambda abstractions are not type annotated). Biernacki and Polesiuk [6] considered the coherence problem of coercion semantics. Their criterion for coherence of the translation is *contextual equivalence* in the target calculus. Inspired by this approach, Bi et al. [5] proposed the canonicity relation to prove coherence for a calculus with disjoint intersection types and BCD subtyping. As we have shown in Sect. 5, constructing a suitable logical relation for F_i^+ is challenging. On the one hand, the original approach by Alpuim et al. [2] in F_i does not work any more due to the addition of BCD subtyping. On the other hand, simply combining System F's logical relation with λ_i^+'s canonicity relation does not work as expected, due to the issue of well-foundedness. To solve the problem, we employ immediate substitutions and a restriction to predicative instantiations.

BCD subtyping and decidability. The BCD type system was first introduced by Barendregt et al. [3] to characterize exactly the strongly normalizing terms. The BCD type system features a powerful subtyping relation, which serves as a base for our subtyping relation. The decidability of BCD subtyping has been shown in several works [27,38,41,51]. Laurent [28] formalized the relation in Coq in order to eliminate transitivity cuts from it, but his formalization does not deliver an algorithm. Only recently, Laurent [30] presented a general way of defining a BCD-like subtyping relation extended with generic contravariant/-covariant type constructors that enjoys the "sub-formula property". Our Coq formalization extends the approach used in λ_i^+, which follows a different idea based on Pierce's decision procedure [38], with parametric (disjoint) polymorphism and corresponding distributivity rules. More recently, Muehlboeck and Tate [34] presented a decidable algorithmic system (proved in Coq) with union and intersection types. Similar to F_i^+, their system also has distributive subtyping rules. They also discussed the addition of polymorphism, but left a Coq formalization for future work. In their work they regard intersections of disjoint types (e.g., $\mathsf{String} \,\&\, \mathsf{Int}$) as uninhabitable, which is different from our interpretation. As a consequence, coherence is a non-issue for them.

Intersection types, the merge operator and polymorphism. Forsythe [44] has intersection types and a merge-like operator. However to ensure coherence, various

	$\lambda_{,,}$ [16]	λ_i [36]	$\lambda_\wedge^\vee$ [7]	λ_i^+ [5]	F_i [2]	F_i^+
Disjointness	○	●	○	●	●	●
Unrestricted intersections	●	○	●	●	○	●
BCD subtyping	○	○	●	●	○	●
Polymorphism	○	○	○	○	●	●
Coherence	○	◑	○	●	◑	●
Bottom type	○	○	●	○	○	●

Fig. 12. Summary of intersection calculi (● = yes, ○ = no, ◑ = syntactic coherence)

restrictions were added to limit the use of merges. In Forsythe merges cannot contain more than one function. Castagna et al. [12] proposed a coherent calculus $\lambda\&$ to study overloaded functions. $\lambda\&$ has a special merge operator that works on functions only. Dunfield proposed a calculus [16] (which we call $\lambda_{,,}$) that shows significant expressiveness of type systems with unrestricted intersection types and an (unrestricted) merge operator. However, because of his unrestricted merge operator (allowing $1\,,,\,2$), his calculus lacks coherence. Blaauwbroek's $\lambda_\wedge^\vee$ [7] enriched $\lambda_{,,}$ with BCD subtyping and computational effects, but he did not address coherence. The coherence issue for a calculus similar to $\lambda_{,,}$ was first addressed in λ_i [36] with the notion of disjointness, but at the cost of dropping unrestricted intersections, and a strict notion of coherence (based on α-equivalence). Later Bi et al. [5] improved calculi with disjoint intersection types by removing several restrictions, adopted BCD subtyping and a semantic notion of coherence (based on contextual equivalence) proved using canonicity. The combination of intersection types, a merge operator and parametric polymorphism, while achieving coherence was first studied in F_i [2], which serves as a foundation for F_i^+. However, F_i suffered the same problems as λ_i. Additionally in F_i a bottom type is problematic due to interactions with disjoint polymorphism and the lack of unrestricted intersections. The issues can be illustrated with the well-typed F_i^+ expression $\Lambda(\alpha * \bot).\, \lambda x : \alpha.\, x\,,,\,x$. In this expression the type of $x\,,,\,x$ is $\alpha \,\&\, \alpha$. Such a merge does not violate disjointness because the only types that α can be instantiated with are top-like, and top-like types do not introduce incoherence. In F_i a type variable α can never be disjoint to another type that contains α, but (as the previous expression shows) the addition of a bottom type allows expressions where such (strict) condition does not hold. In this work, we removed those restrictions, extended BCD subtyping with polymorphism, and proposed a more powerful logical relation for proving coherence. Figure 12 summarizes the main differences between the aforementioned calculi.

There are also several other calculi with intersections and polymorphism. Pierce proposed $\mathsf{F}_\wedge$ [39], a calculus combining intersection types and bounded quantification. Pierce translates $\mathsf{F}_\wedge$ to System F extended with products, but he left coherence as a conjecture. More recently, Castagna et al. [14] proposed a polymorphic calculus with set-theoretic type connectives (intersections, unions, negations). But their calculus does not include a merge operator. Castagna and

Lanvin also proposed a gradual type system [13] with intersection and union types, but also without a merge operator.

Row polymorphism and bounded polymorphism. Row polymorphism was originally proposed by Wand [54] as a mechanism to enable type inference for a simple object-oriented language based on recursive records. These ideas were later adopted into type systems for extensible records [19,21,31]. Our merge operator can be seen as a generalization of record extension/concatenation, and selection is also built-in. In contrast to most record calculi, restriction is not a primitive operation in F_i^+, but can be simulated via subtyping. Disjoint quantification can simulate the *lacks* predicate often present in systems with row polymorphism. Recently Morris and McKinna presented a typed language [33], generalizing and abstracting existing systems of row types and row polymorphism. Alpuim et al. [2] informally studied the relationship between row polymorphism and disjoint polymorphism, but it would be interesting to study such relationship more formally. The work of Morris and McKinna may be interesting for such study in that it gives a general framework for row type systems.

Bounded quantification is currently the dominant mechanism in major mainstream object-oriented languages supporting both subtyping and polymorphism. $\mathsf{F}_{<:}$ [10] provides a simple model for bounded quantification, but type-checking in full $\mathsf{F}_{<:}$ is proved to be undecidable [40]. Pierce's thesis [39] discussed the relationship between calculi with simple polymorphism and intersection types and bounded quantification. He observed that there is a way to "encode" many forms of bounded quantification in a system with intersections and pure (unbounded) second-order polymorphism. That encoding can be easily adapted to F_i^+:

$$\forall(\alpha <: A).\, B \triangleq \forall(\alpha * \top).\,([A \,\&\, \alpha/\alpha]B)$$

The idea is to replace bounded quantification by (unrestricted) universal quantification and all occurrences of α by $A \,\&\, \alpha$ in the body. Such an encoding seems to indicate that F_i^+ could be used as a decidable alternative to (full) $\mathsf{F}_{<:}$. It is worthwhile to note that this encoding does not work in F_i because $A \,\&\, \alpha$ is not well-formed (α is not disjoint to A). In other words, the encoding requires unrestricted intersections.

7 Conclusion and Future Work

We have proposed F_i^+, a type-safe and coherent calculus with disjoint intersection types, BCD subtyping and parametric polymorphism. F_i^+ improves the state-of-art of compositional designs, and enables the development of highly modular and reusable programs. One interesting and useful further extension would be implicit polymorphism. For that we want to combine Dunfield and Krishnaswami's approach [17] with our bidirectional type system. We would also like to study the parametricity of F_i^+. As we have seen in Sect. 5.2, it is not at all obvious how to extend the standard logical relation of System F to account for disjointness, and avoid potential circularity due to impredicativity. A promising solution is to use step-indexed logical relations [1].

Acknowledgments. We thank the anonymous reviewers and Yaoda Zhou for their helpful comments. This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17258816, and by the Research Foundation - Flanders.

References

1. Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 69–83. Springer, Heidelberg (2006). https://doi.org/10.1007/11693024_6
2. Alpuim, J., Oliveira, B.C.d.S., Shi, Z.: Disjoint polymorphism. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 1–28. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_1
3. Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A filter lambda model and the completeness of type assignment. J. Symb. Logic **48**(04), 931–940 (1983)
4. Bi, X., Oliveira, B.C.d.S.: Typed first-class traits. In: European Conference on Object-Oriented Programming (ECOOP) (2018)
5. Bi, X., Oliveira, B.C.d.S., Schrijvers, T.: The essence of nested composition. In: European Conference on Object-Oriented Programming (ECOOP) (2018)
6. Biernacki, D., Polesiuk, P.: Logical relations for coherence of effect subtyping. In: International Conference on Typed Lambda Calculi and Applications (TLCA) (2015)
7. Blaauwbroek, L.: On the interaction between unrestricted union and intersection types and computational effects. Master's thesis, Technical University Eindhoven (2017)
8. Breazu-Tannen, V., Coquand, T., Gunter, C.A., Scedrov, A.: Inheritance as implicit coercion. Inf. Comput. **93**(1), 172–221 (1991)
9. Brent, R.P., Kung, H.T.: The chip complexity of binary arithmetic. In: Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing, pp. 190–200 (1980)
10. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. **17**(4), 471–523 (1985)
11. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(05), 509 (2009)
12. Castagna, G., Ghelli, G., Longo, G.: A calculus for overloaded functions with subtyping. In: Conference on LISP and Functional Programming (1992)
13. Castagna, G., Lanvin, V.: Gradual typing with union and intersection types. In: Proceedings of the ACM on Programming Languages, vol. 1, no. (ICFP), pp. 1–28 (2017)
14. Castagna, G., Nguyen, K., Xu, Z., Im, H., Lenglet, S., Padovani, L.: Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In: Principles of Programming Languages (POPL) (2014)
15. Curien, P.L., Ghelli, G.: Coherence of subsumption, minimum typing and typechecking in $f_{\leq}$. Math. Struct. Comput. Sci. (MSCS) **2**(01), 55 (1992)
16. Dunfield, J.: Elaborating intersection and union types. J. Funct. Program. (JFP) **24**(2–3), 133–165 (2014)
17. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. In: International Conference on Functional Programming (ICFP) (2013)

18. Fokkinga, M.M.: Tupling and mutumorphisms. Squiggolist **1**(4) (1989)
19. Gaster, B.R., Jones, M.P.: A polymorphic type system for extensible records and variants. Technical report, University of Nottingham (1996)
20. Gibbons, J., Wu, N.: Folding domain-specific languages: deep and shallow embeddings (functional pearl). In: ICFP, pp. 339–347. ACM (2014)
21. Harper, R., Pierce, B.: A record calculus based on symmetric concatenation. In: Principles of Programming Languages (POPL) (1991)
22. Henglein, F.: Dynamic typing: syntax and proof theory. Sci. Comput. Program. **22**(3), 197–230 (1994)
23. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. High.-Order Symb. Comput. **23**(2), 167 (2010)
24. Hindley, R.: The principal type-scheme of an object in combinatory logic. Trans. Am. Math. Soc. **146**, 29–60 (1969)
25. Hinze, R.: An algebra of scans. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 186–210. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27764-4_11
26. Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic embedding of DSLs. In: International Conference on Generative Programming and Component Engineering (GPCE) (2008)
27. Kurata, T., Takahashi, M.: Decidable properties of intersection type systems. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 297–311. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0014060
28. Laurent, O.: Intersection types with subtyping by means of cut elimination. Fundam. Inf. **121**(1–4), 203–226 (2012)
29. Laurent, O.: A syntactic introduction to intersection types (2012, unpublished note)
30. Laurent, O.: Intersection subtyping with constructors. In: Proceedings of the Ninth Workshop on Intersection Types and Related Systems (2018)
31. Leijen, D.: Extensible records with scoped labels. Trends Funct. Program. **5**, 297–312 (2005)
32. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978)
33. Morris, J.G., McKinna, J.: Abstracting extensible data types. In: Principles of Programming Languages (POPL) (2019)
34. Muehlboeck, F., Tate, R.: Empowering union and intersection types with integrated subtyping. In: OOPSLA (2018)
35. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 2–27. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31057-7_2
36. Oliveira, B.C.d.S., Shi, Z., Alpuim, J.: Disjoint intersection types. In: International Conference on Functional Programming (ICFP) (2016)
37. Oliveira, B.C.d.S., van der Storm, T., Loh, A., Cook, W.R.: Feature-oriented programming with object algebras. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 27–51. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39038-8_2
38. Pierce, B.C.: A decision procedure for the subtype relation on intersection types with bounded variables. Technical report, Carnegie Mellon University (1989)
39. Pierce, B.C.: Programming with intersection types and bounded polymorphism. Ph.D. thesis, University of Pennsylvania (1991)
40. Pierce, B.C.: Bounded quantification is undecidable. Inf. Comput. **112**(1), 131–165 (1994)

41. Rehof, J., Urzyczyn, P.: Finite combinatory logic with intersection types. In: Ong, L. (ed.) TLCA 2011. LNCS, vol. 6690, pp. 169–183. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21691-6_15
42. Rendel, T., Brachthäuser, J.I., Ostermann, K.: From object algebras to attribute grammars. In: Object-Oriented Programming, Systems Languages and Applications (OOPSLA) (2014)
43. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Proceedings of the IFIP 9th World Computer Congress (1983)
44. Reynolds, J.C.: Preliminary design of the programming language Forsythe. Technical report, Carnegie Mellon University (1988)
45. Reynolds, J.C.: The coherence of languages with intersection types. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 675–700. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54415-1_70
46. Oliveira, B.C.d.S., Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006, Nottingham, United Kingdom, 19–21 April 2006, pp. 199–216 (2006)
47. Schwinghammer, J.: Coherence of subsumption for monadic types. J. Funct. Program. (JFP) **19**(02), 157 (2008)
48. Scott, D.: Outline of a mathematical theory of computation. Oxford University Computing Laboratory, Programming Research Group (1970)
49. Scott, D.S., Strachey, C.: Toward a Mathematical Semantics for Computer Languages, vol. 1. Oxford University Computing Laboratory, Programming Research Group (1971)
50. Siek, J., Thiemann, P., Wadler, P.: Blame and coercion: together again for the first time. In: Conference on Programming Language Design and Implementation (PLDI) (2015)
51. Statman, R.: A finite model property for intersection types. Electron. Proc. Theor. Comput. Sci. **177**, 1–9 (2015)
52. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1989 (1989)
53. Wadler, P.: The expression problem. Java-Genericity Mailing List (1998)
54. Wand, M.: Complete type inference for simple objects. In: Symposium on Logic in Computer Science (LICS) (1987)
55. Zhang, W., Oliveira, B.C.d.S: Shallow EDLs and object-oriented programming. Program. J. (2019, to appear)