Compositional Programming

WEIXIN ZHANG, University of Bristol, United Kingdom and The University of Hong Kong, China
YAOZHU SUN, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

Modularity is a key concern in programming. However, programming languages remain limited in terms of modularity and extensibility. Small canonical problems, such as the Expression Problem (EP), illustrate some of the basic issues: the dilemma between choosing one kind of extensibility over another one in most programming languages. Other problems, such as how to express dependencies in a modular way, add up to the basic issues and remain a significant challenge.

This paper presents a new statically typed modular programming style called *Compositional Programming*. In Compositional Programming, there is no EP: it is easy to get extensibility in multiple dimensions (i.e. it is easy to add new variants as well as new operations). Compositional Programming offers an alternative way to model data structures that differs from both algebraic datatypes in functional programming and conventional OOP class hierarchies. We introduce four key concepts for Compositional Programming: *compositional interfaces*, *compositional traits*, *method patterns*, and *nested trait composition*. Altogether these concepts allow us to naturally solve challenges such as the Expression Problem, model attribute-grammar-like programs, and generally deal with modular programs with *complex dependencies*. We present a language design, called CP, which is proved to be type-safe, together with several examples and three case studies.

CCS Concepts: • **Software and its engineering → Object oriented languages**.

Additional Key Words and Phrases: Expression Problem, Compositionality, Traits

ACM Reference Format:
Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 9 (September 2021), 61 pages. https://doi.org/10.1145/3460228

1 INTRODUCTION

Modularity is a key concern in programming. Programming languages support the development of modular programs by providing language constructs for modularization. For instance, many languages support some notion of modules that can group various kinds of definitions and functions, and can be separately compiled. On a smaller scale, most Object-Oriented Programming (OOP) languages support *classes* which can also be separately defined, compiled, and reused by subclassing.

An important aspect of programming is how to define data structures and the operations over those data structures. Different language designs offer different mechanisms for this purpose. OOP languages model data structures using class hierarchies and techniques such as the Composite pattern [Gamma et al. 1994]. Typically, there is an interface that specifies all the operations (methods) of interest for the data structure. Multiple classes implement different types of nodes in the data

Authors' addresses: Weixin Zhang, University of Bristol, Bristol, United Kingdom, The University of Hong Kong, Hong Kong, China, wxzhang2@cs.hku.hk; Yaozhu Sun, The University of Hong Kong, Hong Kong, China, yzsun@cs.hku.hk; Bruno C. d. S. Oliveira, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk.

0164-0925/2021/9-ART9 $15.00
https://doi.org/10.1145/3460228

structure, supporting all the operations in the interface. Many functional languages employ *algebraic datatypes* [Burstall et al. 1981] to model data structures, and use functions (typically defined by pattern matching) to model the operations over those data structures. For example, in Haskell, we can model a simple form of arithmetic expressions and their evaluation as:

```
data Exp where              -- Datatype definition called Exp
  Lit :: Int -> Exp         -- Constructor
  Add :: Exp -> Exp -> Exp  -- Constructor

eval :: Exp -> Int          -- Evaluation function defined by pattern matching
eval (Lit n)      = n
eval (Add e1 e2)  = eval e1 + eval e2
```

The datatype definition `Exp` defines a (binary) tree structure that models arithmetic expressions with two *constructors* for numeric literals (`Lit`) and additions (`Add`). The `eval` function, defines evaluation by pattern matching over values of `Exp` and calling itself recursively on the child nodes.

As widely acknowledged by the Expression Problem (EP) [Wadler 1998], both algebraic datatypes and class hierarchies have modularity problems. With algebraic datatypes and pattern matching, adding new operations is easy, but adding constructors is hard. Conversely, the OOP approach makes it easy to add new classes, but makes it hard to add new methods at the same time.

This paper presents a new statically typed modular programming style called *Compositional Programming*. In Compositional Programming, there is no EP: it is easy to get extensibility in two dimensions (i.e. it is easy to add new constructors as well as new operations). Compositional Programming offers an alternative way to model data structures that differs from both algebraic datatypes in functional programming and conventional OOP class hierarchies. Thus, compositional Programming can be viewed as an alternative programming paradigm, since programs are structured differently from traditional FP and OOP. The key ideas of Compositional Programming are implemented in a new programming language called CP. For example, the code for modeling arithmetic expressions can be expressed in CP as:

```
type ExpSig<Exp> = {        -- Compositional interface for expressions with a sort Exp
  Lit : Int -> Exp;         -- Constructor (returns the sort Exp)
  Add : Exp -> Exp -> Exp;  -- Constructor (returns the sort Exp)
};

type Eval = { eval : Int };          -- Concrete type for evaluation (instantiates Exp)
evalNum = trait implements ExpSig<Eval> => {      -- Compositional trait for evaluation
  (Lit     n).eval = n;                           -- Definition using a method pattern
  (Add e1 e2).eval = e1.eval + e2.eval;           -- Definition using a method pattern
};
```

In the CP code above, the definition `ExpSig<Exp>` (a compositional interface) plays a similar role to the algebraic datatype `Exp` in functional languages. The special type parameter `Exp` in `ExpSig<Exp>` is called a *sort*, and models types that represent datatypes in CP. All constructors in CP must have a return type which is a sort. Like in functional languages, in CP adding new operations is easy using a special form of *traits* [Schärli et al. 2003]. Unlike functional programming, where adding new constructors to `Exp` is difficult and non-modular, in CP the addition of new constructors is also easy. We will later illustrate the modularity of CP in detail. In essence, there are four new key concepts in CP:

- **Compositional interfaces** can be viewed as an extension of traditional OOP interfaces, such as those found in languages like Java or Scala. In addition to declaring method signatures, compositional interfaces also allow specification of the *constructor signatures*. In turn, this enables *programming the construction of objects against an interface*, instead of a concrete implementation. Compositional interfaces can be parametrized by sorts (such as `Exp`), which abstract over concrete datatypes, and are used to determine which kind of object is built.
- **Compositional traits** extend traditional traits [Schärli et al. 2003] and *first-class traits* [Bi and Oliveira 2018]. Compositional traits allow not only the definition of virtual methods but also the definition of *virtual constructors*. They also allow nested traits, which are used to support *trait families*. Trait families are akin to class families in *family polymorphism* [Ernst 2001]. Both virtual constructor definitions and nested traits are not possible with Schärli et al. [2003]'s traits and previous models of first-class traits.
- **Method patterns**, such as `(Lit n).eval`, provide a lightweight syntax to define method implementations for nested traits, which arise from virtual constructors. This enables compact method definitions for trait families, which resembles programs defined by pattern matching in functional languages, and programs written with attributes in *attribute grammars* [Knuth 1968, 1990].
- **Nested trait composition** is the mechanism used to compose compositional traits. The foundations for this mechanism originate from nested composition, which has been investigated in recent calculi with disjoint intersection types and polymorphism [Alpuim et al. 2017; Bi et al. 2018, 2019; Oliveira et al. 2016]. Our work shows how nested composition can smoothly be integrated into compositional traits. Nested trait composition plays a similar role to traditional class inheritance, but generalizes to the composition of nested traits. Thus it enables a form of inheritance of whole hierarchies, similar to the forms of composition found in family polymorphism. Nested trait composition is *associative* and *commutative* [Bi et al. 2018], just like the composition for traditional traits [Schärli et al. 2003].

Altogether, these concepts allow us to solve various challenges naturally. For instance, they enable a very natural and simple solution to the EP. More interestingly, Compositional Programming can deal with harder modularity challenges, such as modeling interesting classes of attribute grammars or more complex programs, which often contain non-trivial dependencies.

Dependencies are an important concern for modularity: the weaker dependencies between program parts are, the more modular a program is. Realistic programs will depend on some other program parts. A common case of program dependencies is using library functions or functions defined in other parts of the program. A program may also depend on existing *types* and *constructors*. For instance, in OOP, when using a class defined in another part of the program, we may need to refer to the class type and the constructor of the class to create an instance of the class. Unfortunately, in most languages, most dependencies introduce *strong coupling* between various program parts. A simple example of a program with a dependency that introduces strong coupling in functional programming would be a simple pretty-printer for expressions, which depends on `eval`. In a functional language, it is straightforward to write such a program in a non-modular way:

```
printChild :: Exp -> String
printChild (Lit n)     = show n
printChild (Add e1 e2) = if (eval e2 == 0) -- dependency on eval
                         then printChild e1
                         else "(" ++ printChild e1 ++ "+" ++ printChild e2 ")"
```

Such definition of `printChild` is tightly coupled with the concrete implementation of `eval`. Moreover, both `eval` and `printChild` are non-extensible. For modularization to be effective, it is desirable to

have *weak dependencies*. In a modular setting, the definition of `printChild` should depend only on the interface of `eval`, without sticking to a particular implementation. Also, both `eval` and `printChild` should be open to the addition of new cases.

In Compositional Programming, programs with weak dependencies can be expressed concisely, in a statically typed and safe way, and without giving up modularity. More generally, Compositional Programming offers a range of mechanisms to deal with modular programs with various *complex forms of dependencies*. In many existing solutions to the EP, dealing with such dependencies in a modular way is highly non-trivial or simply not possible. There are various *design patterns* [Gamma et al. 1994] partly addressing the problem of strong coupling by using existing programming language features. For example, the Abstract Factory pattern [Gamma et al. 1994], *Object Algebras* [Oliveira and Cook 2012], *Polymorphic Embeddings* [Hofer et al. 2008], *Finally Tagless* [Carette et al. 2009], *Datatypes à la Carte* [Swierstra 2008] or the *Cake Pattern* [Odersky and Zenger 2005] provide ways to abstract over the construction of objects or datatypes. However, such patterns often result in heavily parametrized and boilerplate code. Moreover, the lack of sufficiently powerful composition mechanisms and mechanisms to express weaker forms of dependencies makes it hard to deal with dependencies in such design patterns [Oliveira et al. 2013; Zhang and Oliveira 2017].

The CP language is inspired by the SEDEL language [Bi and Oliveira 2018]. The main novelties over SEDEL are the four compositional programming mechanisms listed above: compositional interfaces, compositional traits, method patterns, and nested trait composition. Compositional Programming is partly inspired by *generalized Object Algebras* [Oliveira et al. 2013], but the built-in language mechanisms make modular programming natural, fully statically typed, and without boilerplate code and excessive parametrization. We present several examples and three case studies in CP. We also introduce a technique called *polymorphic contexts* to deal with components that require some form of context in a modular way. In turn, polymorphic contexts are helpful to model L-attributed grammars. Our first case study is on the design of an Embedded Domain-Specific Language (EDSL) for circuits [Gibbons and Wu 2014; Hinze 2004]. This EDSL is interesting because it has various extensions that can be modularly defined, as well as various dependencies between components. Our second case study is a mini interpreter, which is a larger study and can be extended in several ways. The last case study is an implementation of the C0 compiler, inspired by the work of Rendel et al. [2014]. In this case study, various extensions can be formulated as attributes and those attributes contain non-trivial dependencies to other attributes.

Finally, we present a small calculus that captures the essence of CP. This calculus is shown to be type-safe via an elaboration to the F_i^+ calculus [Bi et al. 2019], which is a recently proposed calculus that supports *disjoint intersection types* [Oliveira et al. 2016], *disjoint polymorphism* [Alpuim et al. 2017] and *nested composition* [Bi et al. 2018].

In summary, the contributions of our work are:

- **Compositional Programming:** We propose a new programming style that encourages weaker dependencies and increases the modularity of programs. Compositional Programming eliminates the EP and can deal with modular programs with complex dependencies.
- **A language design for Compositional Programming:** We present a concrete language design in the form of the CP calculus. The semantics of this calculus is given by elaboration to the F_i^+ calculus and we prove the *type safety* of the elaboration.
- **Attribute Grammars in CP:** We show that CP is powerful enough to implement programs with attributes that are expressible in attribute grammars [Knuth 1968, 1990] in a concise and fully statically type-checked manner. Our technique is partly inspired by the encoding of Rendel et al. [2014], but CP avoids explicit definitions of composition operators, which are necessary in Rendel et al. [2014]'s encoding.

- **Polymorphic contexts:** We introduce a simple technique that combines disjoint polymorphism and Compositional Programming to allow for modular contexts in modular components.
- **Implementation, case studies, and examples:** We have an implementation of CP. We present several examples and three case studies in CP. Altogether, these examples and case studies illustrate how to naturally solve challenges such as the EP or modeling attribute-grammar-like programs. The implementation and case studies can be found in:

https://github.com/wxzh/CP

2 BACKGROUND

This section gives the necessary background to this paper. We first review the well-known Expression Problem (EP) [Wadler 1998] and then move on to the closely related work, namely Object Algebras [Oliveira and Cook 2012] and SEDEL [Bi and Oliveira 2018].

2.1 The Inescapable Expression Problem

In a paper talking about modularity and extensibility, it is hard to avoid the infamous EP, which is a minimal problem illustrating a long-standing extensibility dilemma in programming languages [Wadler 1998]. At the heart of the EP is how to modularly extend a datatype and the operations over it simultaneously. The concrete problem is about how to create a very simple form of expressions (such as numeric literals and addition), and operations over those expressions (such as evaluation and pretty-printing) in a modular way. In OOP, we usually define an abstract class (or interface) for method signatures, and then implement it with various concrete operations:

```
abstract class Exp {
  def eval:  Int
  def print: String
}
class Lit(n: Int) extends Exp {
  def eval  = n
  def print = n.toString
}
class Add(l: Exp, r: Exp) extends Exp {
  def eval  = l.eval + r.eval
  def print = l.print + "+" + r.print
}
```

In this case, it is quite easy to add more data variants (such as multiplication) by creating new classes. However, adding new operations (such as logging) is difficult because every class needs amendments for new methods. Such amendments violate the *open-closed principle* in OOP [Meyer 1988]. The situation is exactly the opposite when it comes to functional programming. As shown by `eval` and `printChild` in Section 1, new operations are just new functions, while adding data variants becomes difficult since it requires modification to every function.

Wadler [1998] and Zenger and Odersky [2005] summarize various requirements that a solution to the EP should satisfy. In short, a solution to the EP should allow modularly adding new forms of expressions and new operations over those expressions, while preserving type-safety and separate compilation. Also desirable is the ability to combine multiple independently developed extensions.

2.2 Object Algebras

Object Algebras [Oliveira and Cook 2012] are a well-known OOP solution to the EP. The abstract syntax of the expressions is described by an Object Algebra interface (a Scala trait):

```
trait ExpAlg[Exp] {
  def Lit: Int       => Exp
  def Add: (Exp,Exp) => Exp
}
```

Essentially `ExpAlg` is an *abstract factory*, where the type parameter `Exp` captures the expression type and capitalized methods are factory methods returning variants of expressions.

Operations over the expressions are *concrete factories* that implement the `ExpAlg` interface. For example, evaluation can be defined as:

```
trait IEval {
  def eval: Int
}
trait Eval extends ExpAlg[IEval] {
  def Lit = n => new IEval {
    def eval = n
  }
  def Add = (e1,e2) => new IEval {
    def eval = e1.eval + e2.eval
  }
}
object eval extends Eval
```

`Eval` implements `ExpAlg` by instantiating `Exp` as `IEval` and defining each factory method accordingly. `Eval` is implemented with a trait instead of a class for allowing mixin composition. To conveniently use `Eval`, an object `eval` is also defined.

Adding new operations. It is easy to add new operations by reimplementing the `ExpAlg` interface. For example, printing is defined in a way similar to evaluation:

```
trait IPrint {
  def print: String
}
trait Print extends ExpAlg[IPrint] {
  def Lit = n => new IPrint {
    def print = n.toString
  }
  def Add = (e1,e2) => new IPrint {
    def print = "(" ++ e1.print ++ "+" ++ e2.print ++ ")"
  }
}
object print extends Print
```

where `Print` instantiates `Exp` as `IPrint` and implements each factory method accordingly.

Adding new variants. It is also easy to add new variants by extending the Object Algebra interface with new factory methods. For example, multiplications are modularly introduced as follows:

```
trait MulAlg[Exp] extends ExpAlg[Exp] {
```

```
  def Mul: (Exp,Exp) => Exp
}
```

where `MulAlg` extends `ExpAlg` with a new factory method `Mul`. Existing operations such as `Eval` can be modularly reused in extensions:

```
trait EvalMul extends MulAlg[IEval] with Eval {
  def Mul = (e1,e2) => new IEval {
    def eval = e1.eval * e2.eval
  }
}
```

where `EvalMul` inherits `Eval` and complements the definition for `Mul` only.

Modular terms. Now we show how to modularly construct a simple addition expression:

```
def exp[Exp](f: ExpAlg[Exp]) = f.Add(f.Lit(0),f.Lit(1))
```

The generic function `exp` builds an addition expression via the factory methods exposed by the abstract algebra `f`. Expressions constructed in this way are modular because concrete expressions can be obtained by calling `exp` with concrete algebras such as `eval` and `print`:

```
println(exp(eval).eval)   // 1
println(exp(print).print) // "(0+1)"
```

By supplying `eval` and `print`, the expression can be respectively evaluated and printed.

2.3 Limitations of Object Algebras

Object Algebras, in their basic form, have several limitations. We discuss the limitations one by one and show how generalized Object Algebras [Oliveira et al. 2013; Rendel et al. 2014] try to address these limitations.

Object Algebra combinators. The first issue is that *there is no proper composition mechanism for Object Algebras.* In the previous section, the expression is indeed constructed *twice* respectively for evaluating and printing. A more efficient way is to compose `Eval` and `Print` into a single algebra so that the expression can be constructed *only once* for both evaluating and printing. A workaround is to use pair-based Object Algebra combinators [Oliveira and Cook 2012], which require explicit projections and hence are inconvenient for use. Fortunately, explicit projections can be eliminated with the help of intersection types. In Scala, the type `A with B` denotes the intersection of two types `A` and `B`, where `A with B` is a subtype of both `A` and `B`. The intersection-based Object Algebra combinator for `ExpAlg` can be defined as:

```
trait ExpMerge[A,B] extends ExpAlg[A with B] {
  val alg1: ExpAlg[A]
  val alg2: ExpAlg[B]
  def lift(x: A, y: B) : A with B

  def Lit = n =>
    lift(alg1.Lit(n), alg2.Lit(n))
  def Add = (e1,e2) =>
    lift(alg1.Add(e1, e2), alg2.Add(e1, e2))
}
```

`ExpMerge` captures the two algebras to be composed as fields `alg1` and `alg2`. `ExpMerge` implements `ExpAlg` by instantiating the type parameter as `A with B` and defining each factory method by firstly

calling the corresponding factory method respectively defined on `alg1` and `alg2` then merging the results via the `lift` method. Concrete composition is done by implementing `ExpMerge` with the values `alg1`, `alg2` and the definition of `lift`. For example, the composition of `Eval` and `Print` is done like this:

```
object evalPrint extends ExpMerge[IEval, IPrint] {
  val alg1 = eval
  val alg2 = print

  def lift(x: IEval, y: IPrint) = new IEval with IPrint {
    def eval  = x.eval
    def print = y.print
  }
}
```

With the composed algebra `evalPrint`, the expression can be constructed only once:

```
val e = exp(evalPrint)
println(e.print + " is " + e.eval) // "(0+1) is 1"
```

Unfortunately, the composition is still done in a cumbersome way. Specialized combinators and a lot of boilerplate code are needed for each Object Algebra interface and for each composition. Workarounds are further proposed, such as using generic combinators and reflection [Oliveira et al. 2013] or a combination of macros and implicits [Rendel et al. 2014].

Dependent operations. The second issue is that *it is hard to model dependent operations*. With conventional Object Algebras, the dependent operation has to be defined together with what it depends on. Recall the pretty-printer that depends on the evaluation discussed in Section 1. It is defined as follows:

```
trait PrintChild extends ExpAlg[IEval with IPrint] {
  def Lit = n => new IEval with IPrint {
    def eval = n
    def print = n.toString
  }
  def Add = (e1,e2) => new IEval with IPrint {
    def eval  = e1.eval + e2.eval
    def print = if (e1.eval == 0) e2.print
                else "(" ++ e1.print ++ "+" ++ ")"
  }
}
```

The type parameter `Exp` is instantiated as `IEval with IPrint` for allowing both `eval` and `print` invocations on expressions. Such an implementation is non-modular because the implementation of `Eval` is repeated inside `PrintChild` and `PrintChild` is tightly coupled with a particular implementation of evaluation.

Generalized Object Algebras. To account for dependencies modularly, a generalization of Object Algebras has been proposed by Oliveira et al. [2013]. The expression Object Algebra interface is generalized by distinguishing negative (input) and positive (output) occurrences of the expression type. For example, `ExpAlg` can be generalized in the following way:

```
trait GExpAlg[Exp,OExp] {
```

```
  def Lit: Int       => OExp
  def Add: (Exp,Exp) => OExp
}
```

where an additional type parameter `OExp` is introduced for capturing positive (output) occurrences of the expression type. Here, the return type positions of the two factory methods are positive and hence are replaced by `OExp`. The ordinary Object Algebra interface can be restored by making `Exp` and `OExp` consistent, which is convenient for defining algebras without dependencies:

```
type ExpAlg[Exp] = GExpAlg[Exp,Exp]
```

By doing this, evaluation and printing algebras can be defined as before. Furthermore, dependent printing can be modularly defined with generalized Object Algebras:

```
trait PrintChild extends GExpAlg[IEval with IPrint, IPrint] {
  def Lit = n => new IPrint {
    def print = n.toString
  }
  def Add = (e1,e2) => new IPrint {
    def print = if (e1.eval == 0) e2.print
                else "(" ++ e1.print ++ "+" ++ ")"
  }
}
```

The dependency on evaluation is expressed by instantiating the input type as `IEval with IPrint` and the output type as `IPrint`. The dependency is modular because `PrintChild` does not depend on a particular implementation of evaluation. The dependency can be fulfilled by composing `PrintChild` with any other algebra that implements `ExpAlg[IEval]` such as `Eval`. However, this requires a generalized Object Algebra combinator for performing such composition.

Summary. The Expression Problem illustrates some fundamental difficulties of writing modular code in current programming languages. Techniques such as Object Algebras provide a solution for such problems, but they have their own limitations. These limitations partly arise from the lack of programming language support. In particular:

- **Unconventional programming style:** The programming style required to program with Object Algebras is quite unconventional compared to standard OOP code. For instance, since constructors are avoided, all objects must be constructed relative to an Object Algebra (or factory), similarly to the method `exp`. Moreover, the code required for programming with Object Algebras is somewhat verbose.
- **No built-in composition:** Scala (or other OOP languages) have built-in support for a form of inheritance, which provides a mechanism to compose code. However, such OOP languages do not support the composition of Object Algebras. Composing Object Algebras in such languages is possible, but requires the explicit definition of composition operators, which have to be defined for each Object Algebra interface.
- **No built-in support for modular dependencies:** Dependencies are quite common in programming. All realistic software will involve multiple forms of dependencies. However, using simple Object Algebras forces dependencies to be written in a non-modular way. Writing modular dependencies is possible with a generalization of Object Algebra interfaces and specialized composition operators. However, such composition operators require a lot of boilerplate code, generalized Object Algebras require a careful manual encoding that

distinguishes positive and negative occurrences, and the programming style involved in such code is generally quite heavyweight and unconventional.

In Section 3, we will show how Compositional Programming addresses these issues and leads to a natural, boilerplate-free, modular programming style.

2.4 SEDEL and First-Class Traits

Intersection types are useful to model modular programs, as we have seen in the previous section. In particular, the Object Algebra combinators use intersection types. However, one useful feature missing in Scala for programming with intersection types is a merge operator. Without this operator, it is sometimes necessary to simulate a merge operator in Scala using meta-programming or reflection [Oliveira et al. 2013; Rendel et al. 2014], which results in convoluted error messages, performance penalties and, more generally, lack of modular type-checking.

Recent developments on disjoint intersection types [Oliveira et al. 2016] provide an alternative approach that supports a native merge operator. The λ_i calculus [Oliveira et al. 2016] was the first calculus with disjoint intersection types and addressed the incoherence problem of intersection types with a merge operator [Dunfield 2014] by introducing the notion of disjointness. The F_i calculus [Alpuim et al. 2017] extends λ_i with a form of parametric polymorphism called *disjoint polymorphism*, where type parameters can be constrained to be disjoint with a specific type. On the other hand, λ_i^+ [Bi et al. 2018] extends λ_i with BCD-style distributive subtyping, enabling nested composition. The F_i^+ calculus [Bi et al. 2019] combines disjoint intersection types, disjoint polymorphism, and nested composition, enabling all the foundational ingredients for Compositional Programming. However, F_i^+ lacks higher-level programming abstractions to make programming convenient. SEDEL [Bi and Oliveira 2018] is a surface language built upon F_i that supports *first-class* traits. That is, unlike Scala traits, SEDEL traits are expressions that can be passed to a function, assigned to a variable, or returned as values. Furthermore, they can be composed to achieve a form of multiple inheritance. Such composition is enabled by the merge operator of F_i.

Trait expressions and self-types. Like Scala traits, SEDEL traits can be annotated with self-types for expressing the dependency on some methods that are implemented by other traits. For example, we can have two mutually dependent traits that respectively implement the methods for testing whether a number is even or odd:

```
type Even = { isEven : Int -> Bool };
type Odd  = { isOdd  : Int -> Bool };

even = trait [self: Odd] => {
  isEven (n : Int) = if n == 0 then true else self.isOdd (n - 1)
} : Even;
odd = trait [self: Even] => {
  isOdd (n : Int) = if n == 0 then false else self.isEven (n - 1)
} : Odd;
```

`Even` and `Odd` are type aliases respectively bound to the record types declaring `isEven` and `isOdd`. By annotating the self-type as `[self: Odd]`, `even` can call `isOdd` via `self` for implementing `isEven` in its body. The `Even` annotation in the end of the trait expression makes sure that `isEven` has been implemented.

Trait types. In the previous example, the type of `even` is `Trait[Odd,Even]` and the type of `odd` is `Trait[Even,Odd]`. Trait types in SEDEL distinguish between the *required* and the *provided* interface. The required interface describes the methods that the trait needs for providing its functionality,

playing a similar role to *abstract methods* in other OOP languages. Meanwhile, the provided interface describes the functionality that the trait offers. For the case of `Trait[Odd,Even]`, `Odd` is the required interface while `Even` is the provided interface. When nothing is required, we can just write `Trait[A]` instead of `Trait[Top,A]`.

Trait instantiations. A trait can be instantiated into an object (a record) using a `new` expression only when its required interface is met. For example, the following attempt to instantiate `even` fails:

```
new[Even] even -- Type Error!
```

Here the object's type, `Even`, must be explicitly specified inside `[]` of the `new` expression. The above instantiation fails because the required interface of `even`, `Odd`, is neither implemented by the trait `even` nor its parents. Nevertheless, `even` can be instantiated together with `odd` since the two traits implement each other's required interface:

```
evenOdd = new[Even & Odd] even & odd; -- OK!
main = evenOdd.isEven 2               --> true
```

The two traits are instantiated into a single object of type `Even & Odd` that implements both `isEven` and `isOdd` methods.

First-class traits, disjoint polymorphism, and dynamic inheritance. An example that differentiates SEDEL's traits from Scala's traits is:

```
combine A [B * A] (x : Trait[A]) (y : Trait[B]) = trait inherits x & y => {};
```

The function `combine` takes two traits, `x` and `y`, and returns a trait that inherits both `x` and `y`. The definition of `combine` illustrates three key features of SEDEL: *disjoint polymorphism*, *first-class traits* and *dynamic inheritance*. Firstly, `combine` is a polymorphic function and the notation `[B * A]`, means that the type parameter `B` is *disjoint* with `A`. This constraint ensures that inheriting `x` and `y` simultaneously will have no conflicts. Secondly, traits are passed as arguments (`x` and `y`) and a trait is the return value of `combine`, showing that traits are *first-class* values. Thirdly, note that what the `trait` expression inherits (`x` and `y`) are parameters of `combine`, which are statically unknown. Such kind of *dynamic inheritance* is not possible in conventional statically typed OOP languages (like Scala or Java), where classes must be statically known, and they cannot be passed as arguments.

Resolving conflicts. Trait composition (or inheritance) in SEDEL follows the traditional trait model [Schärli et al. 2003] where two traits cannot have conflicts for composition to be successful. A benefit of the trait model is that composition is *commutative* and *associative*, which ensures that the order of composition is irrelevant. In the implementation of SEDEL, trait composition is encoded in terms of the merge operator, which is itself *commutative* and *associative* and does not allow for overlapping (or conflicting) values [Bi et al. 2018]. In the case that two traits being composed have conflicts, those conflicts must be explicitly resolved in SEDEL. Suppose that we have two traits `t1` and `t2` that contain conflicting fields:

```
t1 = trait => { f = 1; g = "a" };
t2 = trait => { f = 2; g = "b" };
```

For a trait that would like to inherit from both `t1` and `t2`, the conflicts must be explicitly resolved. Otherwise, a type error will be reported saying that `t1` and `t2` are not disjoint. One way to resolve the conflicts is:

```
t3 = trait [self: Top] inherits t1 \ {f: Int} & t2 \ {g: String} => {
  override f = super.f + (t1 ^ self).f
};
```

The conflicts are resolved by using the *exclusion* operator (\) to remove `f` and `g` respectively from `t1` and `t2`. Together with a self-type annotation `[self: Top]`, the excluded methods from parents can still be accessed via the *forwarding* operator (^) [Bi and Oliveira 2018]. Here, the overridden `f` sums up the inherited `f` from `t2` via **`super`** and the excluded `f` from `t1` via the forwarding expression. We can construct an object from `t3` to test that `f` actually returns 3:

```
main = (new[{f: Int; g: String}] t3).f  --> 3
```

3 AN OVERVIEW OF COMPOSITIONAL PROGRAMMING

This section presents an overview of Compositional Programming. We start by introducing the basic mechanisms of Compositional Programming with the Expression Problem (EP). We then move on to harder modularity issues, such as various forms of dependencies, which arise in modular programs that compute various forms of attributes. All the CP programs presented here can run in our implementation of CP and can be found in the companion material of this paper.

3.1 The Expression Problem with Compositional Programming

In Compositional Programming, there is no EP: it is both easy to add new variants, as well as new operations. Indeed, an explicit goal of Compositional Programming is that programmers do not need to face the tension of choosing one dimension for extensibility. Therefore, Compositional Programming offers an alternative way to model data structures that differs from both algebraic datatypes and conventional OOP class hierarchies. Because of such fundamental differences, and the more modular programming style, we can think of Compositional Programming as an alternative programming paradigm. We introduce the mechanisms used in the programming language CP by solving the EP next.

Compositional interfaces and sorts. In CP, we use a compositional interface to declare a datatype. Compositional interfaces generalize conventional OOP interfaces: we can define not only the signatures of methods but also those of constructors, which is not possible in conventional OOP languages like Java. The compositional interface for basic arithmetic expressions is:

```
type ExpSig<Exp> = {
  Lit : Int -> Exp;
  Add : Exp -> Exp -> Exp;
};
```

There are two kinds of expressions for now: literals and addition. The type parameter `Exp` wrapped in angle brackets is called a *sort* of the compositional interface, working as the type of both kinds of expressions. `Lit` and `Add` are the signatures of the *constructors* for arithmetic expressions. In CP, constructors always start with a capital letter, while all methods start with a lowercase letter. The return type of a constructor must always be a sort, and there can be arbitrarily many sorts in a compositional interface.

Note that sorts in compositional interfaces are handled differently from normal type parameters, which is why they have a special syntax of angle brackets around them. In essence, compositional interfaces are based on several ideas related to generalized Object Algebras (see Section 2.3), and the special treatment for sorts involves, among other things, distinguishing uses of positive and negative occurrences of the type variables. The semantics of sorts is discussed in detail in Section 5. Nonetheless, these subtle semantic differences are essentially only relevant for programs with dependencies, such as those presented in Section 3.2. For now, it suffices to think of sorts as type parameters.

Analogy with algebraic datatypes. Compositional interfaces defining only constructors play a similar role to algebraic datatypes in functional programming. Like algebraic datatypes, they can be used to define data structures by specifying the name of the data structure and the signatures of the constructors. For example, a Haskell counterpart of the compositional interface above is:

```
data Exp where
  Lit :: Int -> Exp
  Add :: Exp -> Exp -> Exp
```

The sort `Exp` of the compositional interface corresponds to the name of the datatype, whereas in both cases the constructors essentially specify the same information: the signatures of the constructors.

Compositional traits. The unit of code reuse in CP is a generalization of *(first-class) traits* [Bi and Oliveira 2018; Schärli et al. 2003]. The generalization stems from the fact that we can define not only method implementations but also *(virtual) trait constructors* within a trait. In CP, the implementation of the `eval` operation for the basic form of expressions can be done in a trait:

```
type Eval = { eval : Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval = n;
  (Add e1 e2).eval = e1.eval + e2.eval;
};
```

This trait implements the compositional interface `ExpSig` with the sort instantiated as `Eval`, corresponding to its actual operation. `Eval` is another example of a compositional interface, which in this case, because it has no sorts, just degenerates into a conventional OOP-style interface with a method `eval` returning `Int`. Within the trait `evalNum`, we implement the `eval` method for `Lit` and `Add` to evaluate the arithmetic expressions.

Method patterns. The *method pattern* `(Lit n).eval` is a lightweight syntax used to define the `eval` method within the trait (which implements the interface `Eval`) that the constructor `Lit` returns. In CP, method patterns are used to make trait definitions concise. Method patterns allow definitions that resemble functional programming definitions by pattern matching, or attributes in attribute grammars. An alternative way of defining constructors would explicitly use first-class traits, which is essentially what method patterns are desugared to:

```
evalNum = trait implements ExpSig<Eval> => {
  Lit     n = trait => { eval = n; };
  Add e1 e2 = trait => { eval = e1.eval + e2.eval; };
};
```

Nested traits and trait families. As shown above, method patterns inside a trait are essentially defining *nested traits*. The outer trait, `evalNum`, is called a *trait family*. The terminology *trait family* is borrowed from *family polymorphism* [Ernst 2001]. In family polymorphism, a family is a class that contains nested virtual classes. In CP, a trait family is a trait that contains nested virtual traits.

Virtual constructors. One significant difference from most existing languages is that CP's constructors are *virtual*. In languages like Java, there are virtual methods, whose concrete implementation is unknown at the time a class is defined. In this way, the references to methods are loose and determined only when objects are instantiated. However, languages like Java do not support virtual constructors or *virtual classes* [Madsen and Moller-Pedersen 1989]. A *virtual constructor* is not

bound to a specific implementation of a trait. Instead, it conforms to the signature in the compositional interface. With virtual constructors, it is possible to create a trait that constructs an expression *without sticking to a particular implementation* of the compositional interface:

```
expAdd Exp = trait [self : ExpSig<Exp>] => {
  test = new Add (new Lit 4) (new Lit 8);
};
```

In CP, term parameters start with a lowercase letter while type parameters are capitalized, so `Exp` is a type parameter, serving as the sort of `ExpSig`. Thus `expAdd` contains an abstract expression that is not associated with any concrete method implementations.

Self-type annotations. The `expAdd` trait above is parameterized by `Exp` and specifies its self-type as `ExpSig<Exp>`. Such self-type annotations are similar to those in Scala [Odersky et al. 2004], which enable a modular way of injecting dependencies on other operations/constructors. The self-type annotation of `expAdd` expresses that `expAdd` must be merged with some trait that concretely implements `ExpSig` for instantiation. By specifying the self-type, the constructors declared inside `ExpSig`, i.e. `Lit` and `Add`, are directly available for building an expression named `test`. Section 3.2 will discuss additional mechanisms in CP to deal with other forms of dependencies.

Extensibility: adding new operations. Like in functional programming with algebraic datatypes and pattern matching, adding a new operation is trivial in CP. We can just create an independent trait family that implements the compositional interface:

```
type Print = { print : String };
printNum = trait implements ExpSig<Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

The trait `printNum` implements `ExpSig<Print>` and defines `Lit` and `Add` using method patterns, in a way similar to `evalNum`. It modularly supports pretty-printing for expressions.

Nested trait composition. At the heart of Compositional Programming is a powerful composition mechanism called *nested composition* [Bi et al. 2018]. Nested composition allows the composition of multiple trait families. This mechanism can be viewed as a form of multiple (trait) inheritance, but with the added ability to recursively compose nested traits automatically. Thus it provides composition mechanisms that are similar to the ones found in languages with family polymorphism. Like trait composition [Schärli et al. 2003], nested composition in CP is *associative* and *commutative* [Bi et al. 2018]. Furthermore, following the trait model and SEDEL, conflicts arising during composition are rejected. In CP, which is statically typed, such conflicts are statically rejected as type errors, following a similar approach to trait composition in SEDEL. Conflict resolution in CP can be done using method overriding or an explicit exclusion operator (like in many models of traits).

In CP, nested composition is performed by the merge operator `,,` [Dunfield 2014]. For example, to compose the trait families `evalNum` and `printNum` with the aforementioned `expAdd` we can write:

```
e = new evalNum ,, printNum ,, expAdd @(Eval&Print);
```

Note that the merge operator `,,` binds tighter than `new` (but application and type application still bind tighter than `,,`). The type application `expAdd @(Eval&Print)` makes the generic trait `expAdd` concrete by instantiating the type parameter `Exp` as the argument `Eval&Print`. Since the self-type annotation of `expAdd` has been instantiated as `ExpSig<Eval&Print>`, in order to meet this self-type requirement, the trait has to be merged with some trait that simultaneously implements

the `eval` and `print` operations for the constructors exposed by `ExpSig`. The requirement is met by merging `expAdd @(Eval&Print)` with `evalNum` and `printNum`. Thus, the merged trait is successfully instantiated into an object using a **`new`** expression.

It is useful to take a moment and understand why the trait composition performed for `e` and the method calls in the expression above pass type-checking and work as expected. Note that the types of `evalNum`, `printNum` and `expAdd @(Eval&Print)` are, respectively, **`Trait`**`[ExpSig<Eval>]`, **`Trait`**`[ExpSig<Print>]` and **`Trait`**`[ExpSig<Eval&Print>,{test:Eval&Print}]`. Their merge is then of the type **`Trait`**`[ExpSig<Eval&Print>,ExpSig<Eval>&ExpSig<Print>&{test:Eval&Print}]`. The merged trait type is an *instantiatable* trait type (i.e. the provided type is a subtype of the required type) because `ExpSig<Eval>&ExpSig<Print>` is a subtype of `ExpSig<Eval&Print>` in CP. In essence, the subtyping relation employed by CP supports *distributivity* of intersections over other constructs [Bi et al. 2018]. In CP, intersections can distribute over function types, records, and trait types. Through the object `e`, we can both evaluate and print the addition expression:

```
e.test.print ++ " is " ++ e.test.eval.toString  --> "(4+8) is 12"
```

Extensibility: adding new variants. Finally, we show how to add multiplications to the language modularly. Note that this is where Compositional Programming differs from algebraic datatypes and definitions by pattern matching, where such extensions cannot be modularly added. We first define a compositional interface that extends `ExpSig` with a `Mul` constructor:

```
type MulSig<Exp> extends ExpSig<Exp> = {
  Mul : Exp -> Exp -> Exp;
};
```

We then implement evaluation and printing by defining two trait families `evalMul` and `printMul` that respectively inherit `evalNum` and `printNum` and complement a definition for `Mul`:

```
evalMul = trait implements MulSig<Eval> inherits evalNum => {
  (Mul e1 e2).eval = e1.eval * e2.eval;
};
printMul = trait implements MulSig<Print> inherits printNum => {
  (Mul e1 e2).print = "(" ++ e1.print ++ "*" ++ e2.print ++ ")";
};
```

Without editing any existing code, we modularly add new data variants. Note that here we use a new keyword **`inherits`**. In CP, inheritance is based on the merge operator but given a more convenient syntax that resembles conventional OOP. Besides nested composition, **`inherits`** additionally allows fields defined in the parent trait to be overridden but still can be used via **`super`** calls in the trait body. With **`super`** calls and inheritance, we can, for instance, construct a slightly more complex expression:

```
expMul Exp = trait [self : MulSig<Exp>] inherits expAdd @Exp => {
  override test = new Mul super.test (new Lit 4);
};

e' = new evalMul ,, printMul ,, expMul @(Eval&Print);
e'.test.print ++ " is " ++ e'.test.eval.toString  --> "((4+8)*4) is 48"
```

The trait `expMul` inherits `expAdd`, refines the self-type to `MulSig` and overrides the `test` field. The overridden expression reuses the inherited expression via **`super`**`.test`. Finally, the object `e'` supports all of the three constructors together with the evaluation and printing operations.

Summary. For the basic EP, there are already many solutions in the literature, including some in mainstream programming languages [Garrigue 2000; Oliveira and Cook 2012; Oliveira et al. 2006a; Swierstra 2008; Torgersen 2004; Zenger and Odersky 2005]. One interesting aspect of the solution presented here is that it is quite elegant and natural, while solutions in mainstream languages tend to have significant amounts of boilerplate code, or are written in highly parametrized code that is not programmer-friendly. Additionally, the more interesting aspect of Compositional Programming is its wide support for various forms of modular code with dependencies, which is shown next.

3.2 Dependencies and S-Attributed Grammars

In the original EP by Wadler [1998], there are very few *dependencies*. In particular, the operations (`eval` and `print`) depend only on themselves, but they do not depend on other operations. Matters become significantly more complicated in the presence of more advanced forms of *dependencies*, and very few existing solutions to the EP have effective mechanisms to deal with dependencies in a modular way. Two primary mechanisms are used to deal with dependencies in CP:

- **Compositional interface type refinement:** When a trait implements some compositional interface, it can refine the *sort types in input positions*. This allows the child nodes in a data structure to assume some functionality that is not implemented in the enclosing trait, but will eventually be part of the final composition later.
- **Self-type annotations:** Like in Scala, the types of self-references can be specified/refined. This enables the self-references to assume functionality that will be implemented by a different trait that will eventually be composed with the current trait. In the context of trait families, there are two kinds of *self-references*: *object self-references* and *family self-references* [Oliveira et al. 2013]. The trait `expAdd` in Section 3.1 already illustrates family self-references, so in what follows we illustrate only object self-references.

We use examples inspired from *S-attributed grammars* [Knuth 1968] and the work from Rendel et al. [2014] to illustrate how Compositional Programming addresses programs with different kinds of dependencies in a modular way. S-attributed grammars deal with synthesized attributes, which are computed from the children. Compositional Programming also allows dependencies on self. The simplest form of dependencies is dependencies on the same attribute of children, which occurred several times in Section 3.1. Therefore, we focus on two kinds of non-trivial dependencies on other synthesized attributes, which hereinafter we call *child dependencies* and *self dependencies*.

Child dependencies. Child dependencies occur when an attribute depends on other synthesized attributes of the children. Here is the `printChild` example from Section 1 in CP:

```
printChild = trait implements ExpSig<Eval % Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

Here `(Add e1 e2).print` depends on `e2.eval`. However, there is no implementation of `eval` in the trait `printChild`. To make this child dependency feasible, we use compositional interface type refinement: we change the concrete interface being implemented to `ExpSig<Eval % Print>` (instead of just using `ExpSig<Print>`). This syntax means that the input type for the expressions in the `Add` constructor (and other constructors, if any, referring to the sort `Exp`) is `Eval&Print`, while the output type is still `Print`. By doing this, we enable the child nodes of `Exp` to depend on an attribute whose implementation is modularly defined somewhere else. Two examples of trait instantiations are:

```
new printChild ,, expAdd @Print                    -- Type Error!
```

```
new printChild ,, evalNum ,, expAdd @(Print&Eval)  -- OK!
```

This first instantiation attempt fails type-checking because it depends on `eval`, which is missing. The second one works since we merge `printChild` with the trait `evalNum` (which implements `Eval`). Importantly, instead of `evalNum`, we could have used any implementation with the same type.

Contrast with inheritance. It is useful to compare the previous example with the more common OOP approach based on inheritance:

```
printInh = trait implements ExpSig<Eval&Print> inherits evalNum => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

The first thing to notice is that the code in the trait body of `printInh` is exactly the same as that in `printChild`. Conforming to the compositional interface instantiated with `Eval&Print`, `printInh` essentially implements both `eval` and `print` instead of only `print`, because the `evalNum` trait family is inherited. Although both approaches work, `printInh` is tightly coupled with the particular implementation of evaluation coming from `evalNum`. In contrast, `printChild` declares a weaker dependency on the abstract interface of `Eval`. We can delay the combination with a concrete implementation of `Eval` until the instantiation phase. In short, `printChild` allows for *weak* child dependencies, which are not coupled with a particular implementation, while preserving strong static type safety. More generally, most uses of inheritance in CP can be converted into code that has weaker dependencies.

Self dependencies. A second interesting case is dependencies on other synthesized attributes of the self-reference. In the following example, the attribute `print` depends on `self.eval`:

```
printSelf = trait implements ExpSig<Eval % Print> => {
  (Lit     n            ).print = n.toString;
  (Add e1 e2 [self:Eval]).print = if self.eval == 0 then "0"
                                  else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

To deal with this dependency without sticking to a particular implementation of `eval`, we add an (object) self-type annotation `[self:Eval]` to use `self.eval` in `Add`. Note that we also need to change the sort instantiation to `ExpSig<Eval % Print>` like the child dependencies, in order to change the self-type of the returning trait correspondingly. The static type-checker will check whether the trait is later merged with another trait that implements `ExpSig<Eval>`. With no compromises on type safety, CP enables modular weak self dependencies on other attributes.

Mutual dependencies. Finally, a more general form of dependencies is *mutual dependencies*, which happen when two attributes are inter-defined, i.e., they depend on each other. Mutual dependencies can involve both child and self dependencies, as illustrated in the following example:

```
type PrintAux = { printAux : String };
printMutual = trait implements ExpSig<PrintAux % Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = e1.printAux ++ "+" ++ e2.printAux;
};
printAux = trait implements ExpSig<Print % PrintAux> => {
  (Lit     n [self:Print]).printAux = self.print;
```

```
  (Add e1 e2 [self:Print]).printAux = "(" ++ self.print ++ ")";
};
```

The two trait families `printMutual` and `printAux` cooperate to omit the outermost parentheses in pretty-printing. We can see that `(Add e1 e2).print` depends on the `printAux` while `printAux` depends on `print`, thus `print` and `printAux` are mutually dependent. CP handles such mutual dependencies modularly. We can combine the traits and use them as before:

```
(new printMutual ,, printAux ,, expAdd @(Print&PrintAux)).test.print  --> "4+8"
```

Summary. Many attribute grammar systems allow the modular definition of attributes, but this is usually done at the cost of modular type-safety. The compositional mechanisms in CP retain the ability of modularly defining attributes from S-attributed grammars or even attributes from self-references, but in a statically type-safe setting. Moreover, the implementations of the attributes are changeable in the final composition. For example, the aforementioned traits `printNum`, `printChild`, `printSelf`, and `printMutual` all implement the `print` method. A programmer can freely pick his favorite implementation to combine with other attributes, such as `eval`.

4 PARAMETRIC POLYMORPHISM AND L-ATTRIBUTED GRAMMARS

In Section 3, we introduced the basic mechanisms for Compositional Programming and illustrated how CP deals with dependencies. One feature that practically all modern languages support is some form of *parametric polymorphism* (or *generics* in OOP languages). A reasonable question to ask is how Compositional Programming interacts with parametric polymorphism. Furthermore, one may wonder if the combination of parametric polymorphism and Compositional Programming enables novel techniques that are useful for programming.

In this section, we explore this question. CP has full support for a form of parametric polymorphism called *disjoint polymorphism* [Alpuim et al. 2017]. We introduce a novel technique called *polymorphic contexts*, which addresses the problem of different modular components requiring different kinds of contextual information. The technique provides encapsulation of contexts: modular components have access to the contextual information they require but do not have access to contextual information used by other components. This technique is useful to model *inherited attributes*, where it is possible to access attributes from parents or siblings. To achieve this, a context should be attached to pass attributes from top to bottom. There is a close relationship between contextual evaluation and *L-attributed grammars* [Knuth 1968; Rendel et al. 2014].

4.1 Contexts and Modular Components

There are plenty of scenarios where we need to add contexts to our code. One of the most common examples is variable binding in an interpreter. Recall that, in Section 3.1, we defined the type of `eval` as `Int`. But this interface is not suitable for an expression language with variable binding. A naive fix is to modify the existing code and add a context to all the trait families:

```
type Eval = { eval : EnvN -> Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval (env:EnvN) = n;
  (Add e1 e2).eval (env:EnvN) = e1.eval env + e2.eval env;
};
evalVar = trait implements VarSig<Eval> => {
  (Let s e1 e2).eval (env:EnvN) = e2.eval (insert @Int s (e1.eval env) env);
  (Var       s).eval (env:EnvN) = lookup @Int s env;
};
```

Note that we add an `EnvN` parameter to `eval`. Since CP's support for type inference is still limited, while `n`, `e1` and `e2` do not require type annotations, we have to annotate `env` with `EnvN` here. `EnvN` is a map from `String` to `Int`, serving as the variable environment, while `insert` and `lookup` are auxiliary functions on maps. For the `Let` expression in `evalVar`, we evaluate `e1` and insert the evaluation result into `env` before evaluating `e2` (the body of the `Let` expression). For the `Var` expression, we look up the variable name to get its value. Although `evalNum` does not need the context for evaluating arithmetic expressions, we still have to pass the context to the recursive calls to make `env` available everywhere.

So far, it seems that everything works well. But what if we add a new context? For example, many interpreters need to pre-define some primitive functions, which are called *intrinsics*. Therefore, we should add an intrinsic environment to store these intrinsic functions. Just like Common Lisp, functions and values do not share the same namespace in this expression language, so these two environments are independent of each other. We need a second parameter for `eval`, which requires modifying all existing code again:

```
type Eval = { eval : EnvN -> EnvF -> Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval (envN:EnvN) (envF:EnvF) = n;
  (Add e1 e2).eval (envN:EnvN) (envF:EnvF) = e1.eval envN envF + e2.eval envN envF;
};
evalVar = trait implements VarSig<Eval> => {
  (Let s e1 e2).eval (envN:EnvN) (envF:EnvF) =
    e2.eval (insert @Int s (e1.eval envN envF) envN) envF;
  (Var       s).eval (envN:EnvN) (envF:EnvF) = lookup @Int s envN;
};
evalFunc = trait implements FuncSig<Eval> => {
  (LetF s f e).eval (envN:EnvN) (envF:EnvF) = e.eval envN (insert @Func s f envF);
  (AppF  s  e).eval (envN:EnvN) (envF:EnvF) = (lookup @Func s envF) (e.eval envN envF);
};
```

Such an approach to adding contextual information has two main problems:

- **It is highly non-modular:** Every time a new context is needed, all the existing code has to be modified. What is worse, we cannot easily modify the type of context if the previous definitions are from a library. In other words, the library author has to anticipate what kind of contexts will be needed in the future, which is impossible.
- **It does not encapsulate contexts:** Interfaces of contexts are fully exposed, even if they are not directly used. For example, `Let` and `Var` do not touch `envF`, while `LetF` and `AppF` do not touch `envN`. Such unnecessary exposure may lead to unexpected modifications to the contexts which ought to be hidden to avoid exploitation by malicious code.

4.2 Polymorphic Contexts

To address the two problems identified in the previous section, we propose a technique that relies on *disjoint polymorphism*, *intersection types* and *nested composition*. This technique enables modular, encapsulated contexts for modular components. Since we cannot anticipate what a context will evolve to in the future, our idea is to make contexts subject to change using parametric polymorphism.

Evaluation with a polymorphic context. Instead of creating an interface for evaluation with specific contexts, we can parametrize the type of the context:

```
type Eval Context = { eval : Context -> Int };
```

Here `Context` is a type parameter. In essence `Eval` becomes a variant of the *Reader* Monad [Wadler 1992], which is commonly used in functional programming. Similarly to Monads, some initial planning is necessary to adapt the existing code to use polymorphic contexts:

```
evalNum Context = trait implements ExpSig<Eval Context> => {
  (Lit     n).eval (ctx:Context) = n;
  (Add e1 e2).eval (ctx:Context) = e1.eval ctx + e2.eval ctx;
};
```

The trait `evalNum` has a type parameter `Context`, which is used as the type of the context in evaluation. Importantly, when implementing `evalNum`, the only thing one can do with `ctx` is to pass it to the children's call on `eval` since `ctx` is the only value of type `Context` in scope. In other words, parametric polymorphism enforces the encapsulation of the context and ensures that the correct context is passed to the evaluation of the children. To illustrate how CP enforces encapsulation of the context, suppose that we try to extract information from the polymorphic context, for example, by trying to look up a variable in the context:

```
evalNum Context = trait implements ExpSig<Eval Context> => {
  (Lit n).eval (ctx:Context) = lookup @Int "foobar" ctx;  -- Type Error!
  -- (Add e1 e2).eval ... is omitted
}
```

This code fails to type check because the type of `ctx` (i.e. `Context`) is not a subtype of `EnvN`, and there is no dynamic casting in CP that can change its type to `EnvN`. In short, if the polymorphic context is completely polymorphic (i.e. its type is just a type variable) then there is not much that can be done with the context except passing it down to recursive calls.

To instantiate `evalNum`, which requires no context on its own, we can specify `Context` as `Top` and pass `()` (the canonical top value) to `eval`:

```
(new evalNum @Top ,, expAdd @(Eval Top)).add.eval ()  --> 12
```

While the code requires an initial modification to be adapted to polymorphic contexts, no additional changes are necessary when future contexts are added to the program. Moreover, a nice quality of the code is that it is written in a direct style. Using more sophisticated solutions, such as Monads, would give additional expressive power, but often at the cost of writing code in a different style (for instance, with the monadic `do` notation).

Adding components with contexts. Let us revisit the variable binding example. In order to add constructs that deal with variables and binders, a context with an `envN` field is needed:

```
type CtxN = { envN : EnvN };
```

Now, we have to write the code for a trait that deals with the evaluation of variables and binders. But what should be the type of context in this case? Using a fully polymorphic context, as we did for literal and addition expressions, will not work, because we need to extract and update information from the environment. Furthermore, using the type `CtxN` directly as the type of the environment is too specific, because it forces the contexts to contain exactly `CtxN` and nothing else. This would prevent modular context evolution. The answer is to use a context with the intersection type `CtxN&Context`:

```
evalVar (Context * CtxN) = trait implements VarSig<Eval (CtxN&Context)> => {
  (Let s e1 e2).eval (ctx:CtxN&Context) =
    e2.eval ({ envN = insert @Int s (e1.eval ctx) ctx.envN } ,, ctx:Context);
```

```
  (Var       s).eval (ctx:CtxN&Context) = lookup @Int s ctx.envN;
};
```

`Context` is a type variable *disjoint* with `CtxN` (expressed as the annotation `Context * CtxN`). This is an example of disjoint polymorphism [Alpuim et al. 2017], which is supported in CP. The disjointness constraint ensures that when the type variable is instantiated to a concrete type, that type cannot share a common supertype with `CtxN`. By using the type `CtxN&Context` as the type of context, we ensure that we can access the environment while being oblivious of other information in the context. Thus the context remains partly polymorphic and adaptable to future extensions while retaining encapsulation and ensuring that the other information in `Context` cannot be altered.

A record update problem. The variable environment should be updated during the evaluation of the `Let` expression, whereas any other information in the context should be retained as well. Note that, the type of `{envN = insert ...}` is `CtxN`, which does not match `CtxN&Context`. So we have to merge it with `(ctx:Context)` to get back the `Context` part. This upcasting is possible because `Context` is a supertype of `CtxN&Context`. The disjointness constraint also ensures that the merge passes type-checking. The code illustrates that in CP we can do a *polymorphic record update* [Cardelli and Mitchell 1991] in the presence of subtyping, which is a notorious problem in many calculi with polymorphic records. For instance, it is well-known that $F_{<:}$ with records (and many other calculi with bounded quantification) cannot solve the polymorphic record update problem. There are only a few calculi with polymorphic records and subtyping that can deal with this problem [Cardelli 1994; Cardelli and Mitchell 1991; Poll 1997], and CP and its core foundation F_i^+ are among them. Note that the problem is simpler *without subtyping*, and various calculi with row polymorphism can address a polymorphic record update (for instance Chlipala [2010]; Leijen [2005], among others).

A second component with a context. To support intrinsic functions, we need a second environment `EnvF`, and a corresponding trait family:

```
type CtxF = { envF : EnvF };
evalFunc (Context * CtxF) = trait implements FuncSig<Eval (CtxF&Context)> => {
  (LetF s f e).eval (ctx:CtxF&Context) =
    e.eval ({ envF = insert @Func s f ctx.envF } ,, ctx:Context);
  (AppF  s  e).eval (ctx:CtxF&Context) = (lookup @Func s ctx.envF) (e.eval ctx);
};
```

Similarly, a polymorphic record update is needed for the `LetF` expression. Although these polymorphic trait families are defined separately, we can still merge them together using nested composition:

```
expPoly Exp = trait [self : ExpSig<Exp>&VarSig<Exp>&FuncSig<Exp>] => {
  test = new LetF "f" (\(x:Int) -> x * x)
                  (new Let "x" (new Lit 9) (new AppF "f" (new Var "x")));
};
e = new evalNum @(CtxN&CtxF) ,, evalVar @CtxF ,, evalFunc @CtxN ,,
        expPoly @(Eval (CtxN&CtxF));
e.test.eval { envN = empty @Int, envF = empty @Func }  --> 81
```

During the composition of `e`, the context types used by other trait families are passed as type arguments to make the final context consistent.

4.3 L-Attributed Grammars

The previous examples only access attributes from parents. It is also possible for inherited attributes to depend on siblings. There is a superset of the aforementioned S-attributed grammars called

L-attributed grammars [Knuth 1968], where inherited attributes depend on parents and left siblings. In such grammars, attributes can be easily evaluated by a left-to-right depth-first traversal.

To illustrate L-attributed grammars, we take pretty-printing as an example. We use a position number to represent each terminal node in this pretty-printing function. The position number is determined by the pre-order traversal of the syntax tree. Before computing it, we need an auxiliary attribute called `cnt` that calculates the total number of nodes in the current subtree:

```
type Cnt = { cnt : Int };
cnt = trait implements ExpSig<Cnt> => {
  (Lit     n).cnt = 1;
  (Add e1 e2).cnt = e1.cnt + e2.cnt + 1;
};
```

With the help of `cnt`, we can compute that $e_1.\mathsf{pos} = e_0.\mathsf{pos} + 1$ and $e_2.\mathsf{pos} = e_0.\mathsf{pos} + e_1.\mathsf{cnt} + 1$, where e_0 is the parent node of e_1 and e_2. The latter equation is a typical example of L-attributed grammars because the attribute depends on both its parent ($e_0.\mathsf{pos}$) and left sibling ($e_1.\mathsf{cnt}$). In our encoding, such computation is done on the parent side and the result is passed down using polymorphic contexts:

```
type Pos = { pos : Int };
type InhPos = { pos1 : Pos -> Int; pos2 : Pos -> Cnt -> Int };
type PrintPos Ctx = { print : Pos&Ctx -> String };

printPos (Ctx * Pos) = trait [self : InhPos] implements ExpSig<Cnt % PrintPos Ctx> => {
  (Lit     n).print (inh:Pos&Ctx) = "{" ++ inh.pos.toString ++ "}";
  (Add e1 e2).print (inh:Pos&Ctx) =
    "(" ++ e1.print ({ pos = pos1 inh } ,, inh:Ctx) ++ "+" ++
           e2.print ({ pos = pos2 inh e1 } ,, inh:Ctx) ++ ")";
  pos1 (e0:Pos) = e0.pos + 1;
  pos2 (e0:Pos) (e1:Cnt) = e0.pos + e1.cnt + 1;
};
```

To compute the inherited attribute `pos`, we introduce two auxiliary functions in `InhPos`. They are implemented in accordance with the equations stated before. The self-type annotation `[self:InhPos]` is added for `(Add e1 e2).print` to access these two functions. Just like previous environments in interpreters, `pos` serves as a part of the context parameter of `print`. Since there is a child dependency on `cnt`, the sort is instantiated as `<Cnt % PrintPos Ctx>`. The position number is calculated and passed down via `print` calls in `Add`, while `inh.pos` is finally used in `Lit`.

With all of the traits ready, we can compose an expression and call the `print` function like before:

```
exp Exp = trait [self : ExpSig<Exp>] => {
  test = new Add (new Add (new Lit 1) (new Lit 2)) (new Add (new Lit 3) (new Lit 4));
};
e = new cnt ,, printPos @Top ,, exp @(Cnt & PrintPos Top);
e.test.print { pos = 0 }  --> (({2}+{3})+({5}+{6}))
```

Since no other contexts need to be mixed together, we just pass `Top` as the type argument of `printPos`. In the last invocation on `print`, we set the initial context to `{pos = 0}`. This means that the root node of the whole syntax tree is marked as 0, then the parent of the left subtree is 1 and the leaves are 2 and 3. Similarly, the parent of the right subtree is 4 and the leaves are 5 and 6. We finally print the position numbers of leaf nodes, as the code shows above.

Program	P	::=	$D; P \mid E$
Declarations	D	::=	$M \mid \textbf{type}\ X\langle\overline{\alpha}\rangle\ \textbf{extends}\ A = B$
Term declarations	M	::=	$x = E \mid (L\ \overline{x : A}\ [\textbf{self} : B]).\ell = E$
Types	A, B	::=	$\textbf{Int} \mid \alpha \mid \top \mid \bot \mid A \to B \mid \forall(\alpha * A).B \mid A\ \&\ B \mid \{\ell : A\}$
		$\mid$	$\textbf{Trait}[A, B] \mid X\langle\overline{S}\rangle$
Sorts	S	::=	$A \mid A\ \%\ B$
Expressions	E	::=	$i \mid x \mid \top \mid \lambda x.E \mid E_1\ E_2 \mid \Lambda(\alpha * A).E \mid E\ @A \mid E_1\ ,,\ E_2 \mid \{\overline{M}\} \mid E.\ell$
		$\mid$	$E : A \mid \textbf{let}\ x : A = E_1\ \textbf{in}\ E_2 \mid \textbf{open}\ E_1\ \textbf{in}\ E_2 \mid \textbf{new}\ E \mid E_1\hat{}E_2$
		$\mid$	$\textbf{trait}[\textbf{self} : A]\ \textbf{implements}\ B\ \textbf{inherits}\ E_1\ \texttt{=>}\ E_2$
Term contexts	Γ	::=	$\bullet \mid \Gamma, x : A$
Type contexts	Δ	::=	$\bullet \mid \Delta, \alpha * A \mid \Delta, X\langle\overline{\alpha}, \overline{\beta}\rangle \mapsto A$
Sort contexts	Σ	::=	$\bullet \mid \Sigma, \alpha \mapsto \beta$

Fig. 1. Syntax of CP.

In fact, our encoding does not only allow L-attributed grammars, any inherited and synthesized attributes can be implemented by contextual evaluation. Nevertheless, L-attributed grammars correspond to a one-pass traversal and ensure termination, so they are the most common usage of attribute grammars and a good example for polymorphic contexts.

Summary. With polymorphic contexts, L-attribute-grammar-like programs are expressed in a statically safe way. Such programs are not uncommon in real-world applications. Interpreters or other operations over ASTs are typical applications where non-trivial forms of attributes can occur. We have shown how variable and intrinsic environments are supported in the previous examples. Besides the two, more contexts may emerge when a language evolves, such as dynamic scoping, mutable parameters, and error handling.

There are three advantages of our approach to polymorphic contexts: 1) it enforces the recursive calls to take the full context as an argument; 2) the polymorphic portion of the context cannot be fiddled with, i.e., the only thing one can do is to pass it unchanged; 3) nonetheless, polymorphic contexts can still be refined for particular uses and expose just the right amount of information while hiding the remaining information. Polymorphic contexts, as well as the ability to express complex forms of attributes, are a valuable supplement to the modularity of Compositional Programming.

5 FORMALIZATION

This section presents the syntax and semantics of CP. The syntax of CP extends SEDEL [Bi and Oliveira 2018] with constructs for Compositional Programming (i.e. compositional interfaces, compositional traits, method patterns, and nested trait composition). The semantics of CP is given by elaborating to a call-by-name formulation of F_i^+ [Bi et al. 2019], a typed calculus combining disjoint intersection types and polymorphism with BCD-style subtyping [Barendregt et al. 1983]. Nested trait composition is built on top of F_i^+'s nested composition [Bi et al. 2018]. We prove that the elaboration to F_i^+ is type-safe and coherent.

5.1 Syntax

Fig. 1 gives the syntax of CP. A program is a sequence of declarations followed by an expression.

Declarations. There are two kinds of declarations: type declarations and term declarations. A type declaration $\textbf{type}\ X\langle\overline{\alpha}\rangle\ \textbf{extends}\ A = B$ is used for two purposes: introducing type aliases

and declaring compositional interfaces. To simplify the formalization, we do not formalize type declarations with conventional type parameters (i.e. we only consider type declarations with sorts). However, adding type parameters can be done in standard ways and we have such declarations in our implementation. The type (or type constructor) X is parametrized with a sequence of sorts $\langle\overline{\alpha}\rangle$ and can optionally extend another type. There are two forms of term declarations: $x = E$ is for simple variable bindings; $(L\ \overline{x : A}\ [\mathbf{self} : B]).\ell = E$ is a *method pattern* serving as syntactic sugar for defining single-field traits conveniently.

Types. Metavariables A and B range over types. Types include integers **Int**, type variables α, the top type $\top$, the bottom type $\bot$, arrows $A \to B$, disjoint quantification $\forall(\alpha * A).B$, intersections $A \,\&\, B$, single-field record types $\{\ell : A\}$ (multi-field record types are syntax sugar for intersections of multiple single-field record types) , trait types $\mathbf{Trait}[A, B]$, and type aliases with sorts instantiated $X\langle\overline{S}\rangle$, where each sort can be either instantiated using a type or a pair of types.

Expressions. Metavariable E ranges over expressions. Expressions include integer literals i, term variables x, the top value $\top$, lambda abstractions $\lambda x.E$, term applications $E_1\ E_2$, type abstractions $\Lambda(\alpha * A).E$, type applications $E\ @A$, merges $E_1\ ,,\ E_2$, multi-field records $\{\overline{M}\}$, record projections $E.\ell$, annotated expressions $E : A$, and (recursive) let bindings $\mathbf{let}\ x : A = E_1\ \mathbf{in}\ E_2$. There are also a few trait related constructs. $\mathbf{trait}[\mathbf{self} : A]\ \mathbf{implements}\ B\ \mathbf{inherits}\ E_1\ \texttt{=>}\ E_2$ specifies an explicit **self** reference of type A, a type B to implement, an inherited trait expression E_1 and a body expression E_2. The **new** E construct instantiates a trait expression. $E_1\hat{}E_2$ is the forwarding expression inherited from SEDEL [Bi and Oliveira 2018]. Inspired by ML-like modules [MacQueen 1984], $\mathbf{open}\ E_1\ \mathbf{in}\ E_2$ is a new construct for directly accessing fields from a record without explicit projections.

5.2 An Informal Introduction to the Elaboration

The semantics of CP is defined by elaborating to F^+_i extended with recursive let bindings. Elaborating a CP program into a F^+_i expression takes several steps, including desugaring, sort transformation, type expansion, etc. The elaboration builds on two ideas from the literature: the denotational model of inheritance by Cook and Palsberg [1989], and generalized Object Algebras [Oliveira et al. 2013]. To better understand the elaboration, let us revisit some of the examples presented in Section 3 and show their concrete elaborations into CP code using more atomic features, such as record types and records, and recursive let bindings.

Elaborating compositional interfaces and sorts. Firstly, the compositional interface `ExpSig`:

```
type ExpSig<Exp> = {
  Lit : Int -> Exp;
  Add : Exp -> Exp -> Exp;
};
```

is translated to an equivalent type using only type parameters:

```
type ExpSig Exp OExp =
  { Lit : Int -> Trait[Exp,OExp] } &
  { Add : Exp -> Exp -> Trait[Exp,OExp] };
```

The sort `Exp` is represented by two type parameters, `Exp` and `OExp`, for respectively capturing the negative and positive occurrences of `Exp`. Negative occurrences of `Exp` (at input positions) are kept unchanged while positive occurrences of `Exp` (at output positions) are changed to `OExp`. For `Add`, the two parameters of type `Exp` are negative. Therefore, only the return type of `Add` is transformed. Since `Add` is a constructor, positive occurrences of `Exp` are further translated to a trait type `Trait[Exp,OExp]`. As a type synonym, `ExpSig` and its right-hand side are put into the type context that tracks type

declarations, among other things. In essence, this transformation is inspired by generalized Object Algebra interfaces [Oliveira et al. 2013] (see also Section 2.3), and the distinction of positive and negative occurrences is helpful for expressing dependencies. If we just wanted to model modular programs *without dependencies*, then distinguishing between positive and negative occurrences would not be necessary.

Similarly, the extended compositional interface `MulSig<Exp>`:

```
type MulSig<Exp> extends ExpSig<Exp> = {
  Mul : Exp -> Exp -> Exp;
};
```

is translated to a type equivalent to:

```
type MulSig Exp OExp =
  { Lit : Int -> Trait[Exp,OExp] } &
  { Add : Exp -> Exp -> Trait[Exp,OExp] } &
  { Mul : Exp -> Exp -> Trait[Exp,OExp] };
```

where the type appearing in the **extends** clause is expanded by looking up the type context and intersecting that type with the type on the original right-hand side.

Elaborating traits. The trait family `evalNum` implements `ExpSig`:

```
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval = n;
  (Add e1 e2).eval = e1.eval + e2.eval;
};
```

which is desugared to:

```
evalNum = trait [self: Top] implements ExpSig Eval Eval => open self in
  { Lit = \(n: Int) -> trait => { eval = n } } ,,
  { Add = \(e1: Eval) -> \(e2: Eval) -> trait => { eval = e1.eval + e2.eval} };
```

The sort instantiation `<Eval>` indicates that there are no dependencies. Therefore, both type parameters (`Exp` and `OExp`) in the elaborated code are instantiated as `Eval`. Since no self-type is specified, the self-reference has type `Top` by default. We can ignore the **open** expression for the moment, since it has no effect in this case, as the self type is `Top`. Method patterns are desugared to functions returning traits and multi-field records are desugared to a merge of multiple single-field records. After type-checking, `evalNum` is further elaborated into a F_i^+ expression:

```
let evalNum = \(self: Top) ->
  { Lit = \(n: Int) -> \(self: Top) -> { eval = n } } ,,
  { Add = \(e1: Eval) -> \(e2: Eval) -> \(self: Top) -> { eval = e1.eval + e2.eval } }
in ...
```

The term declaration is elaborated to a let expression and the trait expressions are elaborated to functions. Since no self-types are specified, the self parameters are all of type `Top`.

Elaborating child dependencies. `printChild` also implements `ExpSig` but contains child dependencies:

```
printChild = trait implements ExpSig<Eval % Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
```

```
};
```

This trait is desugared to:

```
printChild = trait [self: Top] implements ExpSig (Eval&Print) Print => open self in
  { Lit (n: Int) = trait => { print = n.toString } } ,,
  { Add (e1: Eval&Print) (e2: Eval&Print) = trait =>
    { print = if e2.eval == 0 then e1.print
              else "(" ++ e1.print ++ "+" ++ e2.print ++ ")" } };
```

Since `printChild` instantiates the sort as `<Eval % Print>`, `Exp` and `OExp` are respectively instantiated as `Eval&Print` (i.e. the intersection of the two types in `<Eval % Print>`) and `Print` (i.e. the second type in `<Eval % Print>`). Through some type inference, the arguments with expression types in the constructors (such as `e1` and `e2`) are of type `Eval&Print`. This enables `eval` to be called on `e1` and `e2` without an implementation of that operation in the current trait. In short, `printChild` is elaborated to an expression similar to `evalNum` except that expressions arguments are of different type (`Eval&Print`) from the output. Importantly, for the modular definition of `printChild` to work, it is crucial to use different types in the instantiation of `Exp` for input (negative) positions and output (positive) positions.

Elaborating self-type annotations. On the other hand, traits with explicit self-type annotations are desugared differently. For instance, `expAdd`:

```
expAdd Exp = trait [self : ExpSig<Exp>] => {
  test = new Add (new Lit 4) (new Lit 8);
};
```

is desugared to:

```
expAdd = /\Exp. trait [self: ExpSig Exp Exp] => open self in {
  test = new Add (new Lit 4) (new Lit 8);
};
```

The original trait body is wrapped into an **open** expression so that the constructors/methods exposed by the self-type are directly in scope. Further elaboration into F_i^+ results in:

```
let expAdd =
  /\Exp. \(self : { Lit : Int -> Exp -> Exp } & { Add : Exp -> Exp -> Exp -> Exp }) ->
    let Add = self.Add
    in let Lit = self.Lit
    in { test = letrec self : Exp = Add (letrec self : Exp = Lit 4 self in self)
                                        (letrec self : Exp = Lit 8 self in self)
                                        self
               in self }
in ...
```

The type alias used in specifying the self-type is expanded. The type translation rewrites trait types as function types. The **open** `self` expression is elaborated into a series of let bindings, one for each record label. The **new** expressions are elaborated to a lazy fixed point of `self`, following Cook and Palsberg denotational model of inheritance [Cook and Palsberg 1989].

Elaborating inheritance and overriding. We now show the elaboration of a trait that additionally uses **inherits** and **override**. An instance is `expMul`,

```
expMul Exp = trait [self : MulSig<Exp>] inherits expAdd @Exp => {
  override test = new Mul super.test (new Lit 4);
};
```

which is elaborated as:

```
let expMul = /\ Exp. \(self : { Lit : Int -> Exp -> Exp } &
                              { Add : Exp -> Exp -> Exp -> Exp } &
                              { Mul : Exp -> Exp -> Exp -> Exp }) ->
  let super = (expAdd Exp) self
  in (super : Top) ,,
    let Add = self.Add
    in let Lit = self.Lit
    in let Mul = self.Mul
    in { test = letrec self : Exp = Mul super.test
                                        (letrec self : Exp = Lit 4 self in self)
                                        self
               in self }
in ...
```

The inherited trait expression (`expAdd`) is elaborated as a function and applied to the self-reference and the result is bound to **`super`**. Then **`super`** is merged with the body of the trait. Since `test` is overridden, it should be excluded from super; otherwise, a conflict will occur when merging super with the body. The exclusion of `test` from **`super`** is done by injecting a type annotation `Top`.

Elaborating nested composition of traits. Finally, we show how the merge of traits is elaborated using `(new evalNum ,, expAdd @Eval).test.eval` as an example:

```
letrec oself : { Lit : Int -> Top -> {eval : Int} } &
               { Add : {eval : Int} -> {eval : Int} -> Top -> {eval : Int} } &
               { test : {eval : Int} } =
  \(iself : { Lit : Int -> {eval : Int} -> {eval : Int} } &
            { Add : {eval : Int} -> {eval : Int} -> {eval : Int} -> {eval : Int} })
    -> (evalNum iself) ,, ((expAdd {eval : Int} iself) oself)
in oself.test.eval
```

where we have renamed the outer and inner self as `oself` and `iself` for distinction. `evalNum ,, expAdd @Eval` is elaborated to a function that returns the merge of respectively applying `evalNum` and `expAdd @Eval`, which are already elaborated as functions, to `iself`. Since the merged trait is of type `Trait[ExpSig<Eval>,ExpSig<Eval>&{test:Eval}]`, `iself` has the elaborated type of `ExpSig<Eval>` while `oself` has the elaborated type of `ExpSig<Eval>&{test:Eval}`.

5.3 Static Semantics

Overview. Fig. 2 gives an overview of all the relations involved in the elaboration, where $\longrightarrow$ denotes the transformation flow and $\dashrightarrow$ denotes the dependencies between the relations. Firstly, patterns and multi-field records are desugared by $[\![\cdot]\!]$. Then the program P is elaborated into a F_i^+ expression e by $\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e$. During the elaboration process, some transformations and checks are performed: type synonyms and compositional interfaces $X\langle\overline{S}\rangle$ are expanded by $\Delta, \Sigma \vdash A \Rightarrow B$; the right-hand side of a type declaration is transformed by $\Sigma \vdash_p^c A \Rightarrow B$ for distinguishing positive and negative occurrences of sorts; the subtyping relation between two types is checked by $A <: B$; the disjointness of two types is checked by $\Delta \vdash A * B$. Both subtyping and

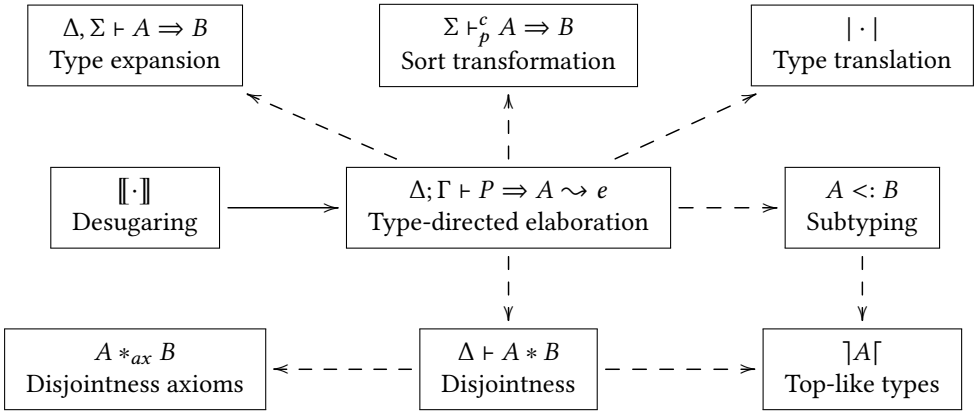

Fig. 2. The relation of relations.

disjointness checking rely on top-like types ($\rceil A \lceil$). The latter further relies on disjointness axioms ($A *_{ax} B$).

Desugaring. The core desugaring rules are:

$$[\![\{M_1, \ldots, M_n\}]\!] \stackrel{\text{def}}{=} [\![\{M_1\}]\!] \,,, \ \ldots \ ,, \, [\![\{M_n\}]\!]$$

$$[\![(L\ \overline{x : A}\ [\mathbf{self} : B]).\ell = E]\!] \stackrel{\text{def}}{=} L = \lambda \overline{x : A}.[\![\mathbf{trait}[\mathbf{self} : B]\ \mathbf{implements}\ \top\ \mathbf{inherits}\ \top \Rightarrow \{\ell = E\}]\!]$$

$$[\![\mathbf{trait}[\mathbf{self} : A]\ \mathbf{implements}\ B\ \mathbf{inherits}\ E_1 \Rightarrow E_2]\!] \stackrel{\text{def}}{=} \mathbf{trait}[\mathbf{self} : A]\ \mathbf{implements}\ B\ \mathbf{inherits}\ [\![E_1]\!] \Rightarrow \mathbf{open\ self\ in}\ [\![E_2]\!]$$

The first rule desugars a multi-field record into an intersection of singleton records. The second rule desugars a method pattern into an ordinary variable binding to a function that returns a single-field trait. The last rule implicitly opens `self` for the trait body so that members declared by the self-type can be directly accessed without prefixing `self`.

Type-checking. Fig. 3 shows the selected typing rules. The gray parts could be ignored for the moment. They will be discussed later in Section 5.5. The type system of CP is bidirectional: under the contexts Δ and Γ, the inference mode ($\Rightarrow$) synthesizes a type A while the checking mode ($\Leftarrow$) checks against A. A lot of the rules are presented previously in the literature [Bi et al. 2019], thus we discuss only the novel ones, which mostly relate to traits and declarations.

The rule T-TyDecl adds a type alias to the type context Δ. The right-hand side, type A, is expanded and transformed before it is added to Δ for type-checking the remaining program. Each sort α has a fresh companion variable β (corresponding to `OExp` in the examples) for distinguishing negative and positive occurrences of sorts. The rule T-TmDecl deals with term declarations. The bound expression, E, is inferred with a type A. Then, x of type A is added to the term context Γ for type-checking the remaining program.

The rule T-Trait is the most complicated one since multiple types and expressions are involved and several validity checks are performed. It firstly expands the self type A and the type to implement B as A_1 and B_1. Then **self** is added to Γ in type-checking the inherited expression E_1 and the body E_2. The inherited expression E_1 is valid only when it is of a trait type $\mathbf{Trait}[A_2, B_2]$ and the requirement A_2 is met by A_1. After validity checking, **super** can be added to Γ in type-checking the body E_2. The type of body should be disjoint to the type of the inherited expression E_1 ($C * B_2$). Meanwhile, the

$$\boxed{\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e}$$

T-TYDECL

$$\frac{\Delta; \overline{\alpha \mapsto \beta} \vdash B \Rightarrow B_1 \qquad \text{fresh } \overline{\beta} \qquad \overline{\alpha \mapsto \beta} \vdash^{\text{false}}_{+} B_1 \Rightarrow B_2 \qquad \Delta; \overline{\alpha \mapsto \beta} \vdash A \Rightarrow A_1 \qquad \Delta, X\langle \overline{\alpha}, \overline{\beta} \rangle \mapsto A_1 \,\&\, B_2; \Gamma \vdash P \Rightarrow C \rightsquigarrow e}{\Delta; \Gamma \vdash \textbf{type } X\langle \overline{\alpha} \rangle \textbf{ extends } A = B; P \Rightarrow C \rightsquigarrow e}$$

T-TMDECL

$$\frac{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e_1 \qquad \Delta; \Gamma, x : A \vdash P \Rightarrow B \rightsquigarrow e_2}{\Delta; \Gamma \vdash x = E; P \Rightarrow B \rightsquigarrow \textbf{let } x : |A| = e_1 \textbf{ in } e_2}$$

$$\boxed{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e}$$

T-TRAIT

$$\frac{\Delta; \bullet \vdash A \Rightarrow A_1 \qquad \Delta; \bullet \vdash B \Rightarrow B_1 \qquad \Delta; \Gamma, \textbf{self} : A_1 \vdash E_1 \Rightarrow \textbf{Trait}[A_2, B_2] \rightsquigarrow e_1 \qquad A_1 <: A_2 \qquad \Delta; \Gamma, \textbf{self} : A_1, \textbf{super} : B_2 \vdash E_2 \Rightarrow C \rightsquigarrow e_2 \qquad C * B_2 \qquad C \,\&\, B_2 <: B_1}{\Delta; \Gamma \vdash \textbf{trait}[\textbf{self} : A] \textbf{ implements } B \textbf{ inherits } E_1 \Rightarrow E_2 \Rightarrow \textbf{Trait}[A_1, C \,\&\, B_2] \rightsquigarrow \lambda(\textbf{self} : |A_1|).\, \textbf{let super} = e_1 \textbf{ self in } e_2 \,,, \textbf{ super}}$$

T-MERGETRAIT

$$\frac{\Delta; \Gamma \vdash E_1 \Rightarrow \textbf{Trait}[A_1, B_1] \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash E_2 \Rightarrow \textbf{Trait}[A_2, B_2] \rightsquigarrow e_2 \qquad \Delta \vdash B_1 * B_2}{\Delta; \Gamma \vdash E_1 \,,, E_2 \Rightarrow \textbf{Trait}[A_1 \,\&\, A_2, B_1 \,\&\, B_2] \rightsquigarrow \lambda(\textbf{self} : |A_1 \,\&\, A_2|).\, e_1 \textbf{ self} \,,, e_2 \textbf{ self}}$$

T-NEW

$$\frac{\Delta; \Gamma \vdash E \Rightarrow \textbf{Trait}[A, B] \rightsquigarrow e \qquad B <: A}{\Delta; \Gamma \vdash \textbf{new } E \Rightarrow B \rightsquigarrow \textbf{let self} : |B| = e \textbf{ self in self}}$$

T-OPEN

$$\frac{\Delta; \Gamma \vdash E_1 \Rightarrow \overline{\{\ell_i : A_i\}} \rightsquigarrow e_1 \qquad \Delta; \Gamma, \overline{\ell_i : A_i} \vdash E_2 \Rightarrow e_2 \rightsquigarrow B \qquad \text{fresh } x}{\Delta; \Gamma \vdash \textbf{open } E_1 \textbf{ in } E_2 \Rightarrow B \rightsquigarrow \textbf{let } x = e_1 \textbf{ in } \overline{\textbf{let } \ell_i : |A| = x.\ell_i} \textbf{ in } e_2}$$

Fig. 3. Selected typing rules.

body and the inherited expression should together implement what B_1 specifies ($C \,\&\, B_2 <: B_1$). The rule T-NEW instantiates a trait. The expression E should be of type $\textbf{Trait}[A, B]$ and the requirement A should be met by B. The rule T-OPEN collects the record types from the inferred type of E_1 and adds every label type pair to the term context in inferring E_2. Besides a rule for ordinary merges, T-MERGETRAIT is a novel rule specially for the merge of two traits. T-MERGETRAIT says that if the two expressions are of type $\textbf{Trait}[A_1, B_1]$ and $\textbf{Trait}[A_2, B_2]$ and B_1 and B_2 are disjoint, then the merged expression is of type $\textbf{Trait}[A_1 \,\&\, A_2, B_1 \,\&\, B_2]$. Inferring the merged traits as a trait type rather than an intersection type brings several advantages, which will be discussed in Section 7.

Sort transformation. Fig. 4 shows the sort transformation, which is a phase that replaces positive occurrences of a sort appearing in a type declaration with some other type. Sort transformation is illustrated in the previous section by elaborating the right-hand side of `ExpSig` and `MulSig`. From the names we cannot distinguish sorts from type parameters and therefore a sort context Σ is needed. How a sort is transformed is further determined by two conditions: p and c. The condition p tracks the positions where sorts appear: $+$ indicates a positive position and $-$ indicates a negative position. Positive and negative positions are treated differently: negative positions are kept unchanged while

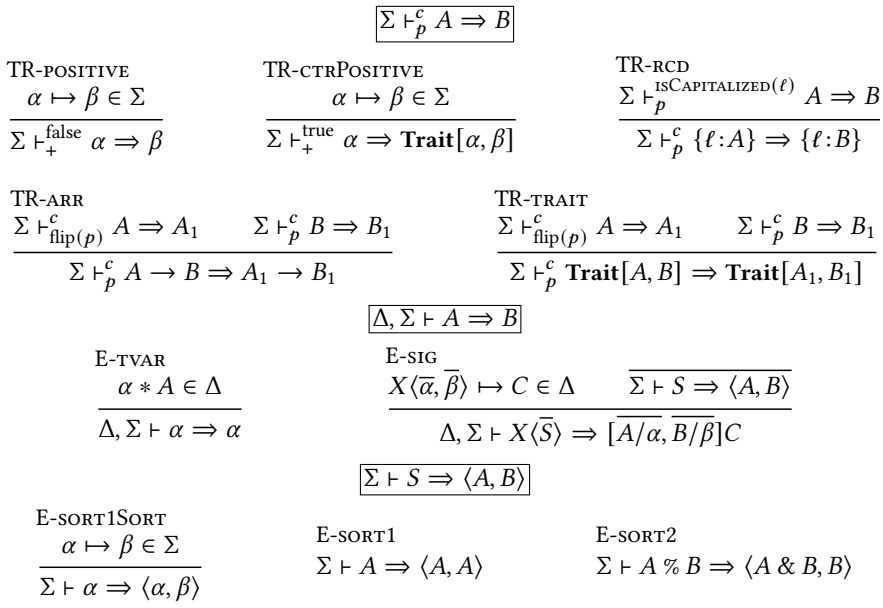

Fig. 4. Selected sort transformation and type expansion rules.

positive positions are transformed. Specifically, for a function type $A \to B$, p is flipped in processing A but unchanged in processing B (rule TR-ARR). The same applies for $\mathbf{Trait}[A, B]$ (rule TR-TRAIT). The boolean value c bookmarks whether the type appears inside a constructor. By default false, c is set to be true when the label of a record is capitalized (rule TR-RCD). According to p and c, rules TR-POSITIVE and TR-CTRPOSITIVE transform the sort α to β and $\mathbf{Trait}[\alpha, \beta]$ respectively.

Type expansion. Type expansion plays two roles: eliminating type aliases and ensuring the well-formedness of the types. It is used, for example, when elaborating the `extends` clause on `MulSig` and `implements` clauses on `evalNum` and `printChild`. Fig. 4 also shows the relevant type expansion rules. The rule E-TVAR checks whether the type variable is in the type context. The rule E-SIG eliminates $X\langle \overline{S} \rangle$ by looking up X in the type context Δ and substituting the negative and positive occurrences of sorts (α and β). An instantiated sort S is expanded to a pair of types for substituting α and β respectively. There are three cases for expanding S: if S is a sort, then α and β are kept abstract (E-SORT1SORT); if S has only one type, both α and β are substituted by that type (E-SORT1); otherwise, α and β are substituted by the intersection of the two types ($A \,\&\, B$) and the second type (B) from the pair (E-SORT2).

Subtyping. Fig. 5 shows the subtyping rules. Most rules come from multiple sources in previous work [Barendregt et al. 1983; Bi and Oliveira 2018; Bi et al. 2018, 2019]. The main novelty is the rule that deal with distributivity of trait types (rules S-DISTTRAIT), which essentially follows the TL-ARR rule for functions (which is inspired by BCD subtyping [Barendregt et al. 1983]). The rule S-DISTTRAIT distributes intersections over the `Trait` type. In the original work on SEDEL, the elaboration targeted the F_i calculus, which is a precursor of F_i^+ without distributivity rules. Thus, that work did not have distributivity for traits. The novel distributivity rules for traits are essential for achieving the nested composition of traits (and trait families). The trait rules and other distributive

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

$$\textsc{S-refl}\ \ A <: A \qquad \textsc{S-trans}\ \ \frac{A <: B \qquad B <: C}{A <: C} \qquad \textsc{S-topLike}\ \ A <: \rceil B \lceil \qquad \textsc{S-bot}\ \ \bot <: A$$

$$\textsc{S-rcd}\ \ \frac{A <: B}{\{\ell : A\} <: \{\ell : B\}} \qquad \textsc{S-andl}\ \ A \,\&\, B <: A \qquad \textsc{S-andr}\ \ A \,\&\, B <: B$$

$$\textsc{S-and}\ \ \frac{A <: B \qquad A <: C}{A <: B \,\&\, C} \qquad \textsc{S-arr}\ \ \frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2} \qquad \textsc{S-forall}\ \ \frac{B_1 <: B_2 \qquad A_2 <: A_1}{\forall(\alpha * A_1).B_1 <: \forall(\alpha * A_2).B_2}$$

$$\textsc{S-trait}\ \ \frac{A_2 <: A_1 \qquad B_1 <: B_2}{\mathbf{Trait}[A_1, B_1] <: \mathbf{Trait}[A_2, B_2]}$$

$$\textsc{S-distArr}\ \ (A \to B) \,\&\, (A \to C) <: A \to B \,\&\, C$$

$$\textsc{S-distTrait}\ \ \mathbf{Trait}[A, B] \,\&\, \mathbf{Trait}[A, C] <: \mathbf{Trait}[A, B \,\&\, C] \qquad \textsc{S-distRcd}\ \ \{\ell : A\} \,\&\, \{\ell : B\} <: \{\ell : A \,\&\, B\}$$

$$\textsc{S-distAll}\ \ \forall(\alpha * A).B \,\&\, \forall(\alpha * A).C <: \forall(\alpha * A).B \,\&\, C$$

Fig. 5. Subtyping.

$$\boxed{\rceil A \lceil}$$

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

Fig. 6. Top-like types

rules allow, for example, `ExpSig<Eval>&ExpSig<Print>` to be subtype of `ExpSig<Eval&Print>`. The rule S-topLike is also novel. It states that any type is a subtype of a top-like type.

Top-like types. Top-like types are types isomorphic to $\top$ (i.e. both sub- and supertypes of $\top$), which was a concept firstly introduced by Barendregt et al. [1983] and then employed by F_i^+ and its precursors [Alpuim et al. 2017; Bi et al. 2019; Oliveira et al. 2016] for proving coherence. We add a new rule TL-trait for trait types, which states that a trait type is top-like when its provided interface is top-like. An important property of top-like types is that they are disjoint to any other types [Alpuim et al. 2017]. Furthermore, the definition of top-like types is crucial for defining disjointness and the inclusion of types such as $\mathbf{Int} \to \top$ in the class of top-like types, which is important to ensure the disjointness of function types and in turn enables merges with multiple functions. A more detailed discussion can be found in work by Huang and Oliveira [2020].

Disjointness. The disjointness judgment detects the conflicts when merging two expressions of type A and B. These rules are omitted here since they are merely a combination of the rules from F_i^+ (which can be found in Fig. 10) and SEDEL. Interested readers can refer to Appendix A for the full disjointness rules of CP.

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \textbf{Int} \mid \alpha \mid \top \mid \bot \mid \tau_1 \to \tau_2 \mid \forall(\alpha * \tau_1).\tau_2 \mid \tau_1 \,\&\, \tau_2 \mid \{\ell : \tau\} \\
\text{Expressions} & e & ::= & i \mid x \mid \top \mid \lambda x.e \mid e_1\ e_2 \mid \Lambda(\alpha * \tau).e \mid e\ \tau \mid e_1\ ,,\ e_2 \mid \{\ell = e\} \mid e.\ell \\
 & & \mid & \textbf{let}\ x : \tau = e_1\ \textbf{in}\ e_2 \\
\text{Term contexts} & \Gamma & ::= & \bullet \mid \Gamma, x : \tau \\
\text{Type contexts} & \Delta & ::= & \bullet \mid \Delta, \alpha * \tau
\end{array}
$$

Fig. 7. F_i^+ syntax extened with (recursive) let bidings.

5.4 The Target Language: F_i^+

The dynamic semantics of CP is given by an elaboration to F_i^+, which is our target language. F_i^+ is a calculus with disjoint intersection types, disjoint polymorphism, and nested composition. The semantics and metatheory (including type-safety and coherence) of F_i^+ have been studied in previous work [Bi et al. 2019]. Here we give an overview of F_i^+, focusing on the typing, subtyping, and disjointness relations, which are the necessary aspects to establish our type-safety theorem in Section 5.5. For more details about the F_i^+ calculus, the interested reader can consult Bi et al. [2019]'s work.

Syntax. Fig. 7 gives the syntax of F_i^+. Metavariable τ ranges over types. Types include integers **Int**, type variables α, the top type $\top$, the bottom type $\bot$, arrows $\tau_1 \to \tau_2$, disjoint quantification $\forall(\alpha * \tau_1).\tau_2$, intersections $\tau_1 \,\&\, \tau_2$, and single-field record types $\{\ell : \tau\}$. Metavariable e ranges over expressions. Expressions include integer literals i, term variables x, the top value $\top$, lambda abstractions $\lambda x.e$, term applications $e_1\ e_2$, type abstractions $\Lambda(\alpha * \tau).e$ with α constrained to be disjoint with τ, type applications $e\ \tau$, merges $e_1\ ,,\ e_2$, single-field records $\{\ell = e\}$, record projections $e.\ell$ and (recursive) let expressions $\textbf{let}\ x : \tau = e_1\ \textbf{in}\ e_2$.

Subtyping. Fig. 8 shows the subtyping rules of the form $\tau_1 <: \tau_2$. The subtyping relation is reflective (TS-refl) and transitive (TS-trans). Rules for top types (TS-top), bottom types (TS-bot), function types (TS-arr) and record types (TS-rcd) are standard. The three rules on intersection types (TS-andL, TS-andR and TS-and) state that $\tau_1 \,\&\, \tau_2$ is the greatest lower bound for τ_1 and τ_2. The BCD-style distributive rules [Barendregt et al. 1983] (TS-distArr, TS-distRcd and TS-distAll) are particularly interesting, since they enable nested composition.

Typing. Fig. 9 gives the bidirectional type system employed by F_i^+: under the contexts Δ and Γ, the inference mode ($\Rightarrow$) synthesizes a type τ from an expression e while the checking mode ($\Leftarrow$) checks the type of an expression against τ. $\vdash \Delta$ and $\Delta \vdash \Gamma$ ensure the well-formedness of contexts. The rule TT-merge infers the merge of two expressions as an intersection type if their types are *disjoint*. The rule TT-tapp additionally checks that the supplied type τ_1 is disjoint with the constraining type τ_2.

Disjointness. The disjointness judgment ($\Delta \vdash \tau_1 * \tau_2$), shown in Fig. 10, ensures that the merge of τ_1 and τ_2 is conflict-free. The disjointness judgment further relies on the definition of top-like types ($\lceil \tau \rceil$) and a disjoint axiom ($\tau_1 *_{ax} \tau_2$). The disjointness axiom contains rules stating that distinct types are disjoint.

Semantics of F_i^+. The semantics of F_i^+ is given by elaborating to F_{co}, a variant of System F extended with products and explicit coercions. Details of F_{co} are beyond the scope of this paper. We refer interested readers to the original F_i^+ paper [Bi et al. 2019]. In the original paper by Bi et al. [2019], F_{co} has a call-by-value semantics. Since F_i^+ is define by elaboration to F_{co} it inherits the call-by-value semantics of F_{co}. Here we assume a call-by-name variant F_{co}, which we expect to be type-sound.

$\boxed{\tau_1 <: \tau_2}$

TS-REFL
$$\tau <: \tau$$

TS-TRANS
$$\frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

TS-TOP
$$\tau <: \top$$

TS-BOT
$$\bot <: \tau$$

TS-RCD
$$\frac{\tau_1 <: \tau_2}{\{\ell : \tau_1\} <: \{\ell : \tau_2\}}$$

TS-ANDL
$$\tau_1 \,\&\, \tau_2 <: \tau_1$$

TS-ANDR
$$\tau_1 \,\&\, \tau_2 <: \tau_2$$

TS-AND
$$\frac{\tau_1 <: \tau_2 \qquad \tau_1 <: \tau_3}{\tau_1 <: \tau_2 \,\&\, \tau_3}$$

TS-ARR
$$\frac{\tau_3 <: \tau_1 \qquad \tau_2 <: \tau_4}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4}$$

TS-TOPARR
$$\top <: \top \to \top$$

TS-TOPRCD
$$\top <: \{\ell : \top\}$$

TS-TOPALL
$$\top <: \forall(\alpha * \top).\top$$

TS-FORALL
$$\frac{\tau_2 <: \tau_4 \qquad \tau_3 <: \tau_1}{\forall(\alpha * \tau_1).\tau_2 <: \forall(\alpha * \tau_3).\tau_4}$$

TS-DISTARR
$$(\tau_1 \to \tau_2) \,\&\, (\tau_1 \to \tau_3) <: \tau_1 \to (\tau_2 \,\&\, \tau_3)$$

TS-DISTRCD
$$\{\ell : \tau_1\} \,\&\, \{\ell : \tau_2\} <: \{\ell : \tau_1 \,\&\, \tau_2\}$$

TS-DISTALL
$$\forall(\alpha * \tau_1).\tau_2 \,\&\, \forall(\alpha * \tau_1).\tau_3 <: \forall(\alpha * \tau_1).\tau_2 \,\&\, \tau_3$$

Fig. 8. F_i^+ subtyping.

$\boxed{\Delta; \Gamma \vdash e \Rightarrow \tau}$

TT-TOP
$$\frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \top \Rightarrow \top}$$

TT-NAT
$$\frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \mathbf{Int}}$$

TT-VAR
$$\frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad (x : A) \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A}$$

TT-APP
$$\frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_2 \Leftarrow \tau_1}{\Delta; \Gamma \vdash e_1\ e_2 \Rightarrow \tau_2}$$

TT-MERGE
$$\frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Delta; \Gamma \vdash e_2 \Rightarrow \tau_2 \qquad \Delta \vdash \tau_1 * \tau_2}{\Delta; \Gamma \vdash e_1\ ,,\ e_2 \Rightarrow \tau_1 \,\&\, \tau_2}$$

TT-ANNO
$$\frac{\Delta; \Gamma \vdash e \Leftarrow \tau}{\Delta; \Gamma \vdash e : \tau \Rightarrow \tau}$$

TT-RCD
$$\frac{\Delta; \Gamma \vdash e \Rightarrow \tau}{\Delta; \Gamma \vdash \{\ell = e\} \Rightarrow \{\ell : \tau\}}$$

TT-PROJ
$$\frac{\Delta; \Gamma \vdash e \Rightarrow \{\ell : \tau\}}{\Delta; \Gamma \vdash e.\ell \Rightarrow \tau}$$

TT-TABS
$$\frac{\Delta, \alpha * \tau_1; \Gamma \vdash e \Rightarrow \tau_2}{\Delta; \Gamma \vdash \Lambda(\alpha * \tau_1).e \Rightarrow \forall(\alpha * \tau_1).\tau_2}$$

TT-TAPP
$$\frac{\Delta; \Gamma \vdash e \Rightarrow \forall(\alpha * \tau_2).\tau_3 \qquad \Delta \vdash \tau_1 * \tau_2}{\Delta; \Gamma \vdash e\ \tau_1 \Rightarrow [\tau_1/\alpha]\tau_3}$$

TT-LET
$$\frac{\Delta; \Gamma, x : A \vdash e_1 \Leftarrow A \qquad \Delta; \Gamma, x : A \vdash e_2 \Rightarrow B}{\Delta; \Gamma \vdash \mathbf{let}\ x : A = e_1\ \mathbf{in}\ e_2 \Rightarrow B}$$

$\boxed{\Delta; \Gamma \vdash e \Leftarrow \tau}$

TT-ABS
$$\frac{\Delta \vdash \tau_1 \qquad \Delta; \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Delta; \Gamma \vdash \lambda x.e \Leftarrow \tau_1 \to \tau_2}$$

TT-SUB
$$\frac{\Delta; \Gamma \vdash e \Rightarrow \tau_2 \qquad \tau_2 <: \tau_1}{\Delta; \Gamma \vdash e \Leftarrow \tau_1}$$

Fig. 9. F_i^+ typing rules.

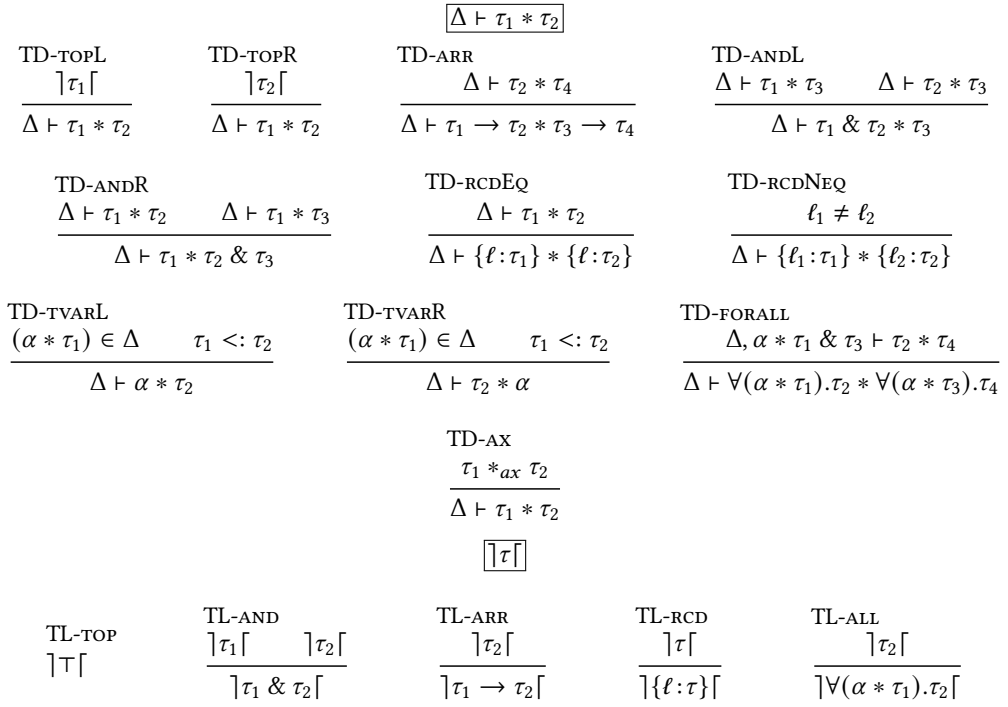

Fig. 10. F_i^+ disjointness.

We expect this result to be straightforward since F_{co} is just a minor variant of System F and System F is known to be type-sound in both call-by-value and call-by-name. The translations from CP to F_i^+ and then to F_{co} are unaffected by the choice of evaluation strategy in F_{co}, and simply inherit the evaluation strategy from F_{co}. As we have mentioned earlier, we also assume lazy recursive let bindings, which are not present in F_i^+. Lazy recursive let bindings are in fact the main motivation to switch to a call-by-name semantics, since they are necessary for the encodings of self references. Although we expect the proof of type-soundness of call-by-name F_{co} and coherence of call-by-name F_i^+ to easily hold, we leave such validation for future work.

5.5 Formal Elaboration

F_i^+ is a subset of CP excluding declarations and trait-related constructs. Therefore, elaborating the shared constructs is straightforward, and only elaborating constructs specific to CP requires some explanations.

Let us focus on the the elaborated F_i^+ expressions (gray parts) shown in Fig. 3. Intuitively, T-tmDecl elaborates a term declaration into a let expression, as illustrated by the elaborations on `evalNum`, `printChild` and `expAdd` in Section 5.2. The gray parts of T-trait and T-new are inherited from SEDEL, which follows Cook and Palsberg [1989]'s denotational semantics of inheritance. Concretely, a trait expression is elaborated to a function that takes **self** as an argument, binds the application result of e_1 **self** to **super** and returns a merge of the body (e_2) and **super**. T-trait is also illustrated by the elaborations on `evalNum`, `printChild` and `expAdd`. As shown by `expAdd`, T-new elaborates the expression into a lazy fixed point of **self**. T-mergeTrait elaborates the merge of two traits into a function that takes **self** as an argument and returns the merge of applying the

two elaborated traits e_1 and e_2 to **self**. Note that the type of the **self** argument for the merged trait is an intersection of the two self types from the two traits being merged. Elaboration on `evalNum ,, expAdd @Eval` illustrates T-MERGETRAIT. T-OPEN elaborates an **open** expression into a sequence of **let** expressions, one for each field from the record e_2. These **let** expressions bind labels to their projections so that e_2 can directly access all the fields from e_1 without explicit projections. The elaboration on `expAdd` is a showcase of T-OPEN.

Finally note that CP's types also need to be translated to F_i^+'s types, because in F_i^+ there is no $\mathbf{Trait}[A, B]$ construct. All types, except $\mathbf{Trait}[A, B]$ have straightforward translations. Trait types are translated to function types in F_i^+ (following SEDEL) with the rule $|\mathbf{Trait}[A, B]| = |A| \to |B|$.

5.6 Type Safety and Coherence

The elaboration of CP into F_i^+ is type-safe and coherent. We summarize the key results here. Detailed proofs and other auxiliary lemmas can be found in Appendix B.

The first result is that elaboration is type-safe. To prove this result we need a few results for some of the auxiliary relations, which are shown next. Note that $|\cdot|$ extends on contexts, denoting that each CP's type in the context is translated to a F_i^+'s type.

LEMMA 5.1 (WELL-FORMEDNESS PRESERVATION). *If* $\Delta, \Sigma \vdash A \Rightarrow B$ *then* $|\Delta| \vdash |B|$.

PROOF. By induction on the derivation of the judgment. □

LEMMA 5.2 (DISJOINTNESS AXIOM PRESERVATION). *If* $A *_{ax} B$ *then* $|A| *_{ax} |B|$.

PROOF. Note that $|\mathbf{Trait}[A, B]| = |A| \to |B|$; the rest are immediate. □

LEMMA 5.3 (SUBTYPING PRESERVATION). *If* $A <: B$ *then* $|A| <: |B|$.

PROOF. By structural induction on the subtyping judgment. □

LEMMA 5.4 (DISJOINTNESS PRESERVATION). *If* $\Delta \vdash A * B$ *then* $|\Delta| \vdash |A| * |B|$.

PROOF. By structural induction on the disjointness judgment. □

Then the main type-safety theorem is:

THEOREM 5.5 (TYPE-SAFETY). *We have that:*

- *If* $\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e$ *then* $|\Delta|; |\Gamma| \vdash e \Rightarrow |A|$.

PROOF. By structural induction on the typing judgment. □

The second theorem is the coherence of the elaboration:

THEOREM 5.6 (COHERENCE). *Each well-typed CP program has a unique elaboration.*

PROOF. For each elaboration rule, the elaborated F_i^+ expression in the conclusion is uniquely determined by the elaborated F_i^+ expressions in the premises. By the coherence property of F_i^+, we conclude that each well-typed CP program has a unique elaboration. Therefore CP is coherent. □

Additional Properties. There are more properties proved in the F_i^+ paper, including the decidability of the type system. These properties should easily hold for CP by extending the proofs with a case for trait types. The cases for trait types are essentially similar to the cases of function types (trait types are actually encoded as function types in the elaboration to F_i^+), so the proof extensions should be straightforward. One thing to notice is that the subtyping relation presented in this paper is non-algorithmic due to the existence of a transitive rule (S-TRANS). An algorithmic variant of the subtyping relation is shown by [Bi et al. 2019]. For proving decidability, we would need to use an extended version of such algorithmic subtyping with trait types.

6 CASE STUDIES

To further demonstrate the applicability of CP, we conducted three case studies. The first case study is a small shallow EDSL for parallel prefix circuits, originally proposed by Hinze [2004] and further studied by Gibbons and Wu [2014]. The second one is a mini interpreter, which integrates and extends the examples in Section 3 and Section 4. The last case study is an implementation of the C0 compiler, inspired by the work of Rendel et al. [2014], which compiles a subset of C to JVM instructions. In all three case studies, the need for dealing with extensibility and complex dependencies arises.

6.1 Scans

Overview. Scans [Hinze 2004] is a DSL for describing parallel prefix circuits. Initially, there are five circuit constructors and an operation on circuits: `Identity` consists of parallel wires; `Fan` connects the leftmost wire to the rest; `Above` is a vertical combinator; `Beside` is a horizontal combinator; `Stretch` inserts identity wires to stretch the circuit; the operation `width` computes the width of a circuit. A few constructs and operations are added later, as extensions: `RStretch` is similar to `Stretch` but stretches the circuit in the opposite direction; `depth` computes the depth of a circuit; `wellSized` checks the well-formedness of a circuit; `layout` compresses the representation of a circuit. Moreover, `wellSized` and `layout` depend on `width`, and `layout` is context-sensitive. Thus, the case study of circuits is interesting because it contains some forms of dependencies. Implementing Scans modularly requires an approach not only to solving the EP but also capable of expressing dependencies. There are already a few implementations written in different languages [Bi et al. 2019; Gibbons and Wu 2014; Zhang and Oliveira 2019]. Still, these implementations are not fully satisfying. We compare our implementation with respect to these implementations. Note, however, that there are no dependencies on self-references in this case study. Thus, this case study does not exercise such a form of dependencies.

Scans in CP. The techniques shown in Section 3 are used in modularizing Scans with CP. The syntax of Scans is captured by a compositional interface:

```
type CircuitSig<Circuit> = {
  Identity : Int -> Circuit;
  Fan      : Int -> Circuit;
  Above    : Circuit -> Circuit -> Circuit;
  Beside   : Circuit -> Circuit -> Circuit;
  Stretch  : (List Int) -> Circuit -> Circuit;
};
```

Operations are modeled as trait families that concretely implement the compositional interface. For example, the implementation of `wellSized` is given below:

```
type WellSized = { wS : Bool };
wellSized = trait implements CircuitSig<Width % WellSized> => {
  (Identity   n).wS = true;
  (Fan        n).wS = true;
  (Above  c1 c2).wS = c1.wS && c2.wS && c1.width == c2.width;
  (Beside c1 c2).wS = c1.wS && c2.wS;
  (Stretch ns c).wS = c.wS && length ns == c.width;
};
```

The trait family `wellSized` implements `CircuitSig<Width % WellSized>`, indicating that it depends on another trait family of `CircuitSig` that implements `Width`. As discussed in Section 3.2, such

dependency is weak and allows us to, for example, call `width` on `c` in `Stretch`. Though the definitions of `layout` and other trait families are omitted here, they can be similarly defined.

New variants. The new constructor `RStretch` is added by extending the compositional interface:

```
type NCircuitSig<Circuit> extends CircuitSig<Circuit> = {
  RStretch : (List Int) -> Circuit -> Circuit
};
```

Accordingly, existing trait families are extended with a case for `RStretch`, for example:

```
nWellSized = trait implements NCircuitSig<Width % WellSized> inherits wellSized => {
  (RStretch ns c).wS = c.wS && length ns == c.width;
};
```

Similarly, `nWidth`, `nDepth`, and `nLayout` extend their respective trait families.

Finally, we obtain the full implementation of Scans by composing all the operations as well as a generic trait that constructs a modular circuit (`circuit` is analogous to `expMul` in Section 3.1):

```
scans = new nWidth ,, nDepth ,, nWellSized ,, nLayout ,,
            circuit @(Width & Depth & WellSized & Layout);
```

Scans in Haskell. Gibbons and Wu [2014] implement Scans as a shallow EDSL in Haskell. Variants of Scans are modeled as functions, thus adding new variants is simple through defining new functions. However, multiple (possibly dependent) operations are defined using tuples. For example, `wellSized` is defined as follows:

```
type Circuit = (Int, Bool)
identity   n = (n, True)
fan        n = (n, True)
above  c1 c2 = (width c1, wellSized c1 && wellSized c2 && width c1 == width c2)
beside c1 c2 = (width c1 + width c2, wellSized c1 && wellSized c2)
stretch ns c = (sum ns, wellSized c && length ns == width c)

width     = fst
wellSized = snd
```

In essence, we have to provide implementations for both `wellSized` and `width` together in a tuple. Such an implementation is not modular because whenever a new operation is needed, existing code has to be modified for accommodating new operations.

A follow-up Haskell implementation [Zhang and Oliveira 2019] employs a technique commonly known as Finally Tagless [Carette et al. 2009], together with a type-class-based encoding of subtyping on tuples, to modularize operation extensions. In essence, such an approach simulates subtyping and inheritance for enabling modular composition of the operations. However, this comes at the cost of boilerplate code and extra complexity due to additional parametrization that is needed to make the encoding work. Moreover, explicit delegations are required for defining dependent operations in the Haskell approach, making the code cumbersome to write. In contrast, CP avoids those issues by having built-in language support for nested composition. Furthermore, compositional interfaces/traits and method patterns make the code quite easy to write.

Scans in Scala. Zhang and Oliveira [2019] present a modular solution in Scala by combining the extensible Interpreter pattern [Wang and Oliveira 2016] and Object Algebras [Oliveira and Cook 2012]. Scans is modeled by a hierarchy of traits, where the root is an interface describing the operations a circuit supports while other traits concretely implement that interface. Adding

new operations is done by defining another trait hierarchy that implements the extended interface and inherits existing traits. Through covariantly refining the circuit fields to the type of the extended interface, previously defined operations can be used as dependencies. Although such an implementation is modular, the extensions are linearly added and the dependencies are strong due to the use of inheritance to express dependencies. Alternatively, the new trait hierarchy can be separately defined without inheriting existing ones. This weakens the dependencies but requires some boilerplate for gluing the hierarchies using mixin composition. For the same example of the multiple interpretation of `Width` and `WellSized`, we need to do extra compositions after defining them separately:

```
trait Identity extends IdentityWidth with IdentityWellSized
trait Fan      extends FanWidth      with FanWellSized
trait Above    extends AboveWidth    with AboveWellSized
trait Beside   extends BesideWidth   with BesideWellSized
trait Stretch  extends StretchWidth  with StretchWellSized
```

Unlike CP, where the composition is done *once* in the family level, *every* trait in Scala needs to be composed because Scala lacks support for nested composition. Another drawback is that Scala's constructors are *not virtual*. Directly calling `new` on constructors for creating objects results in non-modular code. Therefore, Object Algebras [Oliveira and Cook 2012] are used for abstracting over the constructor calls, resulting in more boilerplate.

Scans in F_i^+. Bi et al. [2019] modularize Scans directly in F_i^+ using extensible records. Note that, because Scans do not have self-reference dependencies, there is no strict need for using traits, which are not directly supported in F_i^+. The syntax of Scans is defined similarly to `CircuitSig` except that `Circuit` is captured as a type parameter rather than a sort. The operation extensibility is achieved by defining new record instances while the variant extensibility is achieved by the intersection types and the merge operator. A key difference is how dependent operations are defined:

```
wellSized = {
  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 (ns : List Int) (c : WellSized&Width) =
    { wS = c.wS && length ns == c.width }
};
```

Note that `wellSized` is not given a type. Instead, it defines a record with various fields modeling constructors, and all the "constructor" arguments are explicitly annotated. This illustrates a crucial difference to the CP solution: while in CP `wellSized` (and other operations) implement a proper interface (via `implements`), in the F_i^+ encoding that is not the case. The dependency on `width` is loosely expressed by refining the circuit type as `WellSized&Width`. Such dependency is repeated several times in `above` and `stretch`. The lack of a proper interface when implementing operations allows for a relatively compact solution in F_i^+, but it has some important disadvantages. By implementing an interface in CP, we can ensure various things at the point of the operation definition. For instance, CP checks that: we implement all constructors, and all the constructors are defined with the right number of parameters and types for the parameters. In F_i^+, such checks are not done when defining operations, which is very error-prone. Errors like forgetting to implement a constructor or

	Operation			
Dependency	eval	print	print(aux)	log
Child dependencies		✓		
Self dependencies		✓		✓
Mutual dependencies			✓	
Inherited attributes	✓			

Table 1. Different kinds of dependencies used in the mini interpreter.

implementing a constructor with the wrong number of parameters or the wrong parameter types cannot be checked at the definition point, but are delayed to the composition point.

Evaluation. Besides a qualitative analysis of the aforementioned modular implementations, we further evaluate them in terms of source lines of code (SLOC). To make the comparison fair, we have adapted their implementations to ensure that all the implementations are written in a similar programming style and provide the same functionalities. The SLOC for the modular implementations in Haskell, Scala, F_i^+ and CP are 87, 129, 72 and 70 respectively. CP's solution is the most compact one among the four implementations, while also being the most modular one.

6.2 Mini Interpreter

Overview. The second case study is a mini interpreter for an expression language. This case study is larger (around 700 SLOC) and more comprehensive than the previous one. Besides the EP and simple dependencies, it covers more forms of dependencies. In particular, self dependencies occur in this case study. Furthermore, the case study contains several uses of S-attributed grammars, polymorphic contexts, as well as multiple sorts.

The expression language consists of various sublanguages, including numeric and boolean literals, arithmetic expressions, logical expressions, comparisons, if-then-else branches, variable bindings, function closures, and function applications. The supported operations include a few variants of evaluation, pretty-printing, and logging. The sublanguages are separately defined as features [Prehofer 1997], using different compositional interfaces and trait families. Through nested composition, these features can be arbitrarily combined to form a *product line of interpreters* [Pohl et al. 2005].

Dependencies. The operations on the expression language contain non-trivial dependencies. Table 1 summarizes the different kinds of dependencies used in the mini interpreter. With techniques shown in Section 3.2, these dependencies are expressed in a modular way. For example, `log` is a simple form of logging, which shows the printing and evaluation results of an expression for debugging purposes. Here is a simplified logging implementation for numeric expressions:

```
logNum = trait implements NumSig<(Eval&Print) % Log> => {
  (Add e1 e2 [self:Eval&Print]).log = self.print ++ " is " ++ self.eval.toString;
  -- other constructors are omitted
};
```

To express self dependencies on `eval` and `print`, we annotate `Add`'s self-type as `Eval&Print`.

Polymorphic contexts. There are four different kinds of contexts for evaluation in total: an empty context (`Top`); a map from strings to numbers for evaluating variable bindings; a map for dynamic

scoping; an environment for intrinsic functions. Using techniques shown in Section 4.2, we model these contexts as polymorphic contexts to make the code with contexts modular and evolvable.

Multi-sorted languages. Examples presented in the paper so far are all based on a single-sorted expression language. Essentially, CP allows compositional interfaces to be parameterized by multiple sorts. This ability is demonstrated by the following code excerpt extracted from this case study:

```
type CmpSig<Boolean,Numeric> = {
  Eq  : Numeric -> Numeric -> Boolean;
  Cmp : Numeric -> Numeric -> Numeric;
  -- other constructors are omitted
};
```

`CmpSig` models equality and three-way comparison (also known as the *spaceship* operator), which returns 0 if the two operands are equal, 1 if the left operand is greater, or -1 otherwise. They take `Numeric` arguments and construct `Boolean` and `Numeric` traits respectively. Notice that `CmpSig` is developed as an independent feature. It can later be combined with other independently developed features such as numeric expressions:

```
expCmp N B = trait [self : NumSig<N>&CmpSig<B,N>] => {
  cmp = new Cmp (new Add (new Lit 1) (new Lit 2)) (new Lit 3);
};
```

In `cmp`, we construct an expression with the new constructor `Cmp`, as well as `Lit` and `Add` which are independently developed before.

6.3 C0 Compiler

Overview. The C0 compiler was originally an educational one-pass compiler developed for the compilation course at Aarhus University. Rendel et al. [2014] translated this compiler into their encoding of Object Algebras, whereas we will present this case study in our approach with Compositional Programming. Then we will make a comparison with the implementation by Rendel et al. [2014], as well as the original non-modular version.

C0 selected a subset of the C programming language, consisting of only integer types, arithmetic/bitwise/comparison operations, a few control flow statements, functions, and basic I/O. In other words, it is also a multi-sorted language, whose interfaces are parameterized by `Program`, `Function`, `Statement`, `Expression`, etc. A C0 program consists of function declarations and definitions, which will be compiled into Java bytecode. The original implementation was written in Java and later reimplemented in Scala using Object Algebras by Rendel et al. [2014]. Both implementations include a bytecode generator from AST nodes to JVM instructions as well as a recursive descent parser. Since the CP language is currently a research prototype and does not support I/O or complex string manipulation, we eliminate the parsing phase from this case study. Lexical analysis is not the core part of C0 and will not affect the validity of our evaluation.

Chained attributes. We have shown various forms of dependencies in terms of *S-attributed grammars* in Section 3.2 and introduced polymorphic contexts to tackle *L-attributed grammars* in Section 4.3. However, there are still other kinds of dependencies in attribute grammars. In the C0 compiler, an attribute can depend on both its children and its parent or siblings. For example, consider an attribute that counts the number of leaf nodes (terminal symbols) occurring to the left or in the subtree of the current node. The attribute equations together with the production rules of `Lit` and `Add` are listed below:

$$E \longrightarrow n \quad \{\texttt{Lit}\}$$
$$E.\texttt{count}_s = E.\texttt{count}_i + 1 \qquad (1)$$

$$E_0 \longrightarrow E_1 \text{ "+" } E_2 \quad \{\texttt{Add}\}$$
$$E_1.\texttt{count}_i = E_0.\texttt{count}_i \qquad (2)$$
$$E_2.\texttt{count}_i = E_1.\texttt{count}_s \qquad (3)$$
$$E_0.\texttt{count}_s = E_2.\texttt{count}_s \qquad (4)$$

Note that `count` has two roles: the subscript i stands for *inherited* attributes while s stands for *synthesized* ones.

For terminal symbols, we just add one to the inherited number, as Equation 1 shows. For nonterminal symbols, there are three attribute evaluation rules: Equation 2 is a trivial inherited attribute depending on its parent; Equation 3 is more interesting because it depends on its left sibling; Equation 4 is a trivial synthesized attribute depending on its child. These three equations compose a traversal of the syntax tree: $E_1.\texttt{count}_i$ inherits from its parent $E_0.\texttt{count}_i$ and then does its own computation on the left subtree to obtain $E_1.\texttt{count}_s$; $E_2.\texttt{count}_i$ inherits from its left sibling and then traverses the right subtree to obtain $E_2.\texttt{count}_s$; finally $E_0.\texttt{count}_s$ synthesizes the attribute from its child.

Such a tree traversal reveals an interesting class of attributes called *chained attributes* [Kastens and Waite 1994]. A chained attribute is both inherited and synthesized. If we regard inherited attributes as *Reader* Monads and synthesized as *Writer*, then chained attributes correspond to *State* Monads.

In this case study, `HasVariables` and `HasFunctions` are chained attributes. They are used to do bookkeeping for variable and function declarations. Like inherited attributes, we model them with polymorphic contexts, where the inherited part serves as the context of the corresponding synthesized part:

```
type HasVariables = { variables : Map };  -- inherited
type ChainedVariables Ctx = { variables : HasVariables&Ctx -> Map };  -- synthesized
```

If the chained attribute depends on other attributes, the context can be easily extended as we described in Section 4.2. Here is an example of extending the context with `HasOffset`:

```
type ParamSig<Parameter> = { Param : String -> Parameter };
parameterVariables (Ctx * (HasVariables&HasOffset)) = trait implements
    ParamSig<ChainedVariables (HasOffset&Ctx)> => {
  (Param id).variables (inh : HasOffset&HasVariables&Ctx) (name : String) =
    if name == id then inh.offset else inh.variables name;
};
```

The constructor `Param` is used to declare a parameter within a function, and the trait implements the chained attribute of the variable environment. `(Param id).variables` will update the previous environment `inh.variables` with a new mapping from the given identifier to the current offset which works as the variable index. Note that, although these two `variables` have the same name, the former is a chained attribute while the latter is an inherited attribute. Such a delegation forms a tree traversal to do bookkeeping for parameter declarations.

Comparision. The code statistics of the aforementioned three C0 implementations are shown in Table 2. The original Java implementation inlines semantic actions into the handwritten parser. For the sake of fairness, lexical analysis related lines are not counted and the bytecode prelude is reformatted in the same style as the other two. Although the original code is slightly shorter than CP, it is highly entangled and hinders modularity and extensibility. The operation of code generation is hardcoded in the parsing functions, so it is impossible to add new operations modularly. What is worse, there is no data structure for a syntax tree, leaving no space for extension.

Java (Aarhus University)	SLOC	**Scala** (Rendel et al. [2014])	SLOC	**CP**	SLOC
Entangled Compiler	235	Generic	140	Maybe Algebra	12
(Tokenizer excluded)		Trees, Signatures and Combinators	558	Compositional Interfaces	32
		Composition and Assembly	101		
		Attribute Interfaces	32	Attribute Interfaces	8
		Algebra Implementations	191	Trait Implementations	216
Bytecode (Reformatted)	25	Bytecode Prelude	25	Bytecode Prelude	25
Main	14	Main	5	Main Example	21
Total	274	**Total**	1,052	**Total**	314

Table 2. Source lines of code for the three implementations of the C0 compiler.

To modularize the original C0 implementation, Rendel et al. [2014] use an extended form of generalized Object Algebras to model attribute grammars in Scala. It allows L-attributed grammars with different kinds of dependencies to be modularly defined. Due to the lack of the proper composition mechanisms in Scala, the attributes cannot be easily composed, and specialized composition operators have to be explicitly defined. Such boilerplate code largely accounts for the SLOC reported in "Trees, Signatures and Combinators". In addition, "Composition and Assembly" is the handwritten code to deal with various dependencies. Their "Attribute Interfaces" and "Algebra Implementations" are counterparts of our "Attribute Interfaces" and "Trait Implementations".

Compared to CP, Rendel et al. [2014]'s approach is significantly more verbose (about 3.5x SLOC). In CP, we do not need to write boilerplate code for trait composition and only a few lines of "Compositional Interfaces" are necessary. A workaround they propose to avoid such boilerplate code is to employ meta-programming for generating the specialized signatures and combinators. However, their source code will not be type-checked before macro expansion. Compilation errors will be reported in terms of generated code, which could be confusing for programmers and make it hard to debug. Nevertheless, their assembly mechanism, which relies on specialized combinators that have to be handwritten, encodes the pattern of chained attributes and simplifies the algebra implementations. Our approach of polymorphic contexts is a little more complicated than theirs, but does not require any specialized combinators and works smoothly with nested trait composition.

7 RELATED WORK

Mainstream statically typed OOP and virtual methods. Virtual methods in OOP provide a way to weaken *method dependencies*. In a virtual method call, such as `this.m()` in a method of a class `A`, the call *does not* necessarily refer to the implementation of `m()` in `A`. Instead, it may refer to a later implementation in a subclass of `A`. The choice of the implementation of `this.m()` depends on which subclass of `A` is used to instantiate the object on which `this.m()` is called. However, virtual methods alone are insufficient to weaken other kinds of dependencies. Most programming languages tend to have *static* references to both *constructors* and *types*. For instance, if we refer to a constructor in Java, say `new A()`, then the constructor (unlike the method `this.m()`) *will always refer to the same constructor of class* `A`. Such *static* dependencies create a tight coupling between the use of the constructor and the class `A`, and make programs less modular than they ought to be. Moreover, most statically typed OOP languages use *(static) inheritance* pervasively. Inheritance often creates more coupling than needed between method implementations in subclasses and method implementations in the superclass. In a subclass declaration, such as `class A extends B {...}`, `B` must be some concrete class, with (possibly) some concrete method implementations. In other words, inheritance cannot be parametrized, and we cannot program against the *interface* of the superclass: we must program against some concrete class implementation.

CP adopts virtual methods, while also supporting virtual constructors to avoid static references to constructors. Static references to types, which would typically arise from constructors, are avoided by using sorts. Moreover, in CP, most uses of inheritance can be replaced by code with weaker dependencies (see discussion in Section 3.2), thus avoiding the coupling problems introduced by inheritance. Altogether this leads to code that is more modular and has weaker dependencies than in conventional statically typed OOP.

Mixins and traits. Single inheritance supported by many class-based OOP languages is insufficient for code reuse. On the other hand, multiple inheritance is hard to do correctly due to the existence of the *diamond problem*. Mixins [Bracha and Cook 1990] provide one form of multiple inheritance. The Jigsaw framework [Bracha 1992] formalizes mixins and provides a set of operators on mixins. There are other formalizations of mixins proposed for different languages such as ML-like languages [Ancona and Zucca 2002; Duggan and Sourelis 1996] and Java-like languages [Flatt et al. 1998; Lagorio et al. 2009]. Traits [Schärli et al. 2003] are an alternative to mixins. The fundamental difference between traits and mixins is the way of dealing with conflicts when composing multiple traits/mixins. Mixins *implicitly* resolve the conflicts according to the composition order, whereas the programmer must *explicitly* resolve the conflicts for traits. The trait model avoids unexpected errors caused by the wrong choice of implementation through implicit resolution. Furthermore, it makes trait composition associative and commutative, and the order of traits being composed does not affect semantics (all permutations have the same behavior). This is in contrast with mixins, where the composition is order-sensitive. Typically classes, mixins, and traits in statically typed languages (such as Scala) are second-class and do not support dynamic inheritance. Dynamic languages like JavaScript can encode quite general forms of mixins and support dynamic inheritance. However, type-checking dynamic inheritance is hard. There is little work on typing first-class classes/mixins/traits. Takikawa et al. [2012]'s first-class classes in Typed Racket, Lee et al. [2015]'s tagged objects and Bi and Oliveira [2018]'s first-class traits are three notable works, which support such features in statically typed languages. Our work follows the first-class trait model by Bi and Oliveira [2018]: traits in CP are first-class, statically typed, and support dynamic inheritance.

First-class traits and disjoint intersection types. As discussed in Section 2.4, the work on *disjoint intersection types* [Alpuim et al. 2017; Bi et al. 2018, 2019; Oliveira et al. 2016] provides a suitable foundation for Compositional Programming. In particular, the F_i^+ calculus supports disjoint intersection types, disjoint polymorphism, and nested composition, which are key ingredients for Compositional Programming. However, F_i^+ is still a core calculus and is inconvenient to directly program with. Built upon disjoint intersection types, SEDEL [Bi and Oliveira 2018] is a surface language that adds high-level abstraction mechanisms, particularly on the support for *first-class traits*. However, SEDEL is elaborated to F_i [Alpuim et al. 2017], which is a predecessor of F_i^+ and does not support nested composition. The design of CP improves on the design of SEDEL. The support for *compositional interfaces*, *compositional traits*, *method patterns* and *nested trait composition* are all novel to CP. Thus, the main advantages of CP over SEDEL are the language mechanisms supporting virtual constructors, weak references to datatypes, several new mechanisms to express weaker dependencies (such as compositional interface type refinement), and the ability to compose nested traits via nested composition.

In addition to the new features designed for Compositional Programming, CP further takes advantage of the unrestricted intersections brought by targeting to F_i^+ in improving the trait model proposed by SEDEL. The additional support for nested composition on traits not only unifies the syntax by making the merge operator also work on the `inherits` and `new` clauses (see the difference of the code shown respectively in Section 2.4 and Section 3) but also simplifies the typing rules to

SEDEL. For example, the typing rule for defining traits in SEDEL is:

$$\frac{\overline{\Gamma, \mathbf{self} : B \vdash E_i \Rightarrow \mathbf{Trait}[B_i, C_i] \rightsquigarrow e_i}^{i \in 1..n} \quad \Gamma, \mathbf{self} : B, \mathbf{super} : C_1 \,\&\, .. \,\&\, C_n \vdash \{\overline{\ell_j = E'_j}^{j \in 1..m}\} \Rightarrow C \rightsquigarrow e \quad \overline{B <: B_i}^{i \in 1..n} \quad \Gamma \vdash C_1 \,\&\, .. \,\&\, C_n \,\&\, C \quad C_1 \,\&\, .. \,\&\, C_n \,\&\, C <: A}{\Gamma \vdash \mathbf{trait}[\mathbf{self} : B] \ \mathbf{inherits}\ \overline{E_i}^{i \in 1..n} \{\overline{\ell_j = E'_j}^{j \in 1..m}\} : A \Rightarrow \mathbf{Trait}[B, A] \rightsquigarrow \lambda(\mathbf{self} : |B|).\, \mathbf{let}\ \mathbf{super} = \overline{(e_i\ \mathbf{self})}^{i \in 1..n}\ \mathbf{in}\ \mathbf{super}\ ,,\ e}$$

This rule, among others, is clearly more complicated than its counterpart (T-TRAIT) in CP. The complexity is mainly caused by dealing with expression sequences: every expression needs to be translated and validated. In contrast, CP processes only one expression thanks to the newly introduced rule MERGETRAIT. MERGETRAIT checks the disjointness of two traits and infers their merge as a trait type rather than an intersection type, thus reducing the complexity and duplication. Another important benefit of the CP design is the improved support for *type inference*. For example, in SEDEL, a type must be provided to instantiate a trait, but this type is inferred in CP. Moreover, parameters of a method pattern inside a trait can omit types in CP if they are declared by the type specified in the **implements** clause. This is quite handy for defining trait families.

Virtual classes and family polymorphism. Ideas such as *virtual classes* [Ernst et al. 2006; Madsen and Moller-Pedersen 1989] and *family polymorphism* [Ernst 2001], extend the idea of virtual methods to classes. Thus, *classes* and *constructors* can themselves be *virtual*, weakening the dependencies to classes and constructors. Virtual classes were first introduced in the BETA programming language [Madsen et al. 1993]. BETA supports only single, static inheritance. The composition of BETA programs is done through the fragment system [Knudsen et al. 1994]. The gbeta language [Ernst 1999] extends BETA with mixins and, more importantly, supports family polymorphism [Ernst 2001]. Family polymorphism is a powerful mechanism for extensible and composable software design, which can solve the EP [Ernst 2004]. Clarke et al. [2007] classify family polymorphism approaches into object family approaches and class family approaches. In an object family approach, nested classes are attributes of objects of the family class. Some examples of object family approaches are BETA, gbeta, CaesarJ [Aracic et al. 2006] and Newspeak [Bracha et al. 2010]. Whereas in a class family approach, nested classes are attributes of the family class. Class family approaches include Concord [Jolly et al. 2004], .FJ [Igarashi et al. 2005], Jx [Nystrom et al. 2004], J& [Nystrom et al. 2006] and Familia [Zhang and Myers 2017]. There are also hybrid approaches like Tribe [Clarke et al. 2007]. Object family approaches are typically more expressive but require a more complex dependent type system, e.g. *vc* [Ernst et al. 2006]. The closest approach is Familia [Zhang and Myers 2017], which also supports subtype polymorphism, family polymorphism, and parametric polymorphism but does not support the merge operator.

One difference between CP and the family polymorphism systems is that in those systems, inheritance is still used as a primary mechanism to express dependencies. Similarly to (regular) classes, the use of inheritance in family polymorphism sometimes creates more coupling than necessary between sub- and super-classes/families. In contrast, such dependencies can be weakened via CP's support for self-references and compositional interface type refinement, leading to more modular programs. Another difference is that conflicts are often implicitly resolved based on some order in those systems (e.g., gbeta uses the composition order and Jx uses the dispatch order). In contrast, CP adopts an approach where conflicts are explicitly resolved, following the trait model [Schärli et al. 2003].

SOP, MDSoC, AOP, and FOP. Subject-oriented programming (SOP) [Harrison and Ossher 1993], *multi-dimensional separation of concerns* (MDSoC) [Tarr et al. 1999], *aspect-oriented programming* (AOP) [Kiczales et al. 1997], and *feature-oriented programming* (FOP) [Prehofer 1997] are software paradigms that share a similar vision of *separation of concerns*: i.e., separating a program into different parts so that each part addresses a separate concern. Since in those paradigms, a complete program has been separated, a *composition* mechanism to assemble the parts back together is necessary. Typically SOP, MDSoC, AOP and FOP employ a *meta-programming* approach to software composition. Such meta-programming approaches are usually an extension to an existing programming language, such as Hyper/J and AspectJ for Java. A source-to-source compiler will combine the separated aspects before producing plain Java code. Quite often many of those tools do not fully support modular type-checking or separate compilation.

In contrast, Compositional Programming is a language-based approach, with both clearly defined static and dynamic semantics. The merge operator provides the composition mechanism in Compositional Programming. What distinguishes the elaboration adopted by CP from general meta-programming is that the elaboration is completely transparent for a programmer: 1) Type-checking is done directly in the source language, where type errors (and other well-formedness errors) in programs are reported in terms of the source rather than the target; 2) Type-checking is modular: each definition can be type-checked with only its implementation and type signatures of the dependencies. Worth noting is that the style of elaboration employed by CP to give the semantics to the language is also adopted by other languages. Most notably, the Glasgow Haskell Compiler (GHC) elaborates the source language (Haskell) into a small core language [Sulzmann et al. 2007]. Like CP, all type-checking is done at the source level and the elaboration process is transparent to Haskell programmers. In contrast, in many approaches that employ meta-programming, often there is no source-level type-checking or even some more basic error checking like syntax well-formedness. Consequently, no modular type-checking is offered and errors are reported on the generated program, which are hard to understand.

ML modules. The design of CP is partly inspired by ML module systems [MacQueen 1984]. Analogously to compositional interfaces and trait families in CP, ML signatures and structures can be used to specify and implement the constructors. However, unlike CP, ML modules are neither extensible nor first-class. There are many proposals to extend the ML modules with additional expressiveness, such as making modules first-class [Russo 2000] or recursive [Russo 2001]. Together with other features, ML modules can also be used in solving the EP [Nakata and Garrigue 2006].

Scala. CP is also influenced by Scala [Odersky et al. 2004], where features such as intersection types, traits, and self-types are shared. As compared in Section 2.4, Scala's traits are not first-class and do not support dynamic inheritance and nested composition. Nevertheless, Scala supports virtual types [Igarashi and Pierce 1999; Madsen and Moller-Pedersen 1989], which can be used for simulating family polymorphism but in a much more verbose way [Odersky and Zenger 2005]. In CP, sorts are elaborated to type parameters. Another option is to use virtual types. The strengths and weaknesses of type parameters and virtual types are summarized by Bruce et al. [1998]: type parameters are flexible in composing and decomposing types while virtual types are good at specifying mutually recursive families of classes whose relationships are preserved in the derived family. Type parameters are a natural choice for CP since the underlying F_i^+ calculus supports disjoint polymorphism. In future work, we would like to explore design with virtual types.

Algebraic signatures and Object Algebras. Readers familiar with algebraic signatures [Guttag and Horning 1978] used in algebraic specification languages may observe some similarities to compositional interfaces. Algebraic signatures also allow the definition of constructors. However,

the semantics of constructors in compositional interfaces differs from those in conventional algebraic signatures. The key difference is that the positive and negative occurrences of sorts are distinguished in CP, which is important to support an advanced form of modular dependencies but not well-supported with algebraic signatures.

Oliveira and Cook [2012] explored an encoding of algebraic signatures in OOP languages, yielding a solution to the EP called *Object Algebras*. Other design patterns, such as *Polymorphic Embeddings* [Hofer et al. 2008] and *Finally Tagless* [Carette et al. 2009] use similar encodings. Such encodings originate from the work by Hinze [2006] and Oliveira et al. [2006b]. Hinze [2006] showed how to model Church encodings of GADTs [Cheney and Hinze 2002] using Haskell type classes. Oliveira et al. [2006b] then showed that such type class based encoding can also solve the EP. Object Algebras use parametric interfaces and classes to represent and implement the algebraic signatures, respectively. As discussed in Section 2.3, Object Algebras, in their basic form, are hard to be composed and express dependencies. These limitations are later addressed in Scala by means of *generalized Object Algebras* and intersection types, together with specialized combinators [Oliveira et al. 2013; Rendel et al. 2014]. However, the Scala approaches based on reflection or meta-programming have important drawbacks as discussed in Section 6.3. In contrast, CP has built-in language support for sorts and nested composition, which is more convenient to use and avoids the use of reflection or meta-programming techniques.

Alternatives to context evolution. There are other approaches to context evolution. Monads [Wadler 1992], and in particular Reader Monads, can be helpful for context evolution. Nevertheless, to deal with multiple contexts we would like to have multiple Reader Monads. Unfortunately, this causes some issues in a monadic setting. A notorious problem with Monads is that the composition of two Monads (which would be useful to compose contexts) is not always a Monad. Thus there is no simple way to compose multiple Monads in general. It is possible to use *Monad transformers* [Liang et al. 1995] to compose multiple Monads of different kinds (e.g. a Reader and a Writer Monad, or a Reader and the IO Monad). However, composing Monads of the same kind (like two Reader Monads) is problematic. Only advanced techniques [Jaskelioff 2008; Schrijvers and Oliveira 2011], which require significant sophistication, improve on traditional Monad transformers and can deal smoothly with multiple modular contexts. *Implicit context propagation* [Inostroza and Storm 2015] is another approach in OOP, which requires writing boilerplate code for adapting previous code to accept new contexts (although such boilerplate can be generated). Polymorphic contexts provide us with a simple approach to modularly handle them without changing the existing code, while keeping strong static type safety.

8 CONCLUSION

We have presented key concepts of Compositional Programming together with a language design and implementation called CP. CP's support for compositional interfaces, compositional traits, virtual constructors, and method patterns enables a programming style that allows programs with non-trivial dependencies to be modularized. The applicability of CP is demonstrated by various examples and case studies. The calculus that captures the essence of CP is proved to be type-safe.

We envision Compositional Programming as an alternative programming style and paradigm to what is currently offered in both functional programming and object-oriented programming. Compositional Programming borrows many ideas from both paradigms. The style of Compositional Programming presented in this paper is essentially a purely functional programming style and draws inspiration from languages like Haskell. A purely functional style has benefits in terms of reasoning about programs and it also simplifies some issues related to the composition of code. Multiple inheritance in the presence of mutable state is a notorious problem, for instance. On the

other hand, Compositional Programming also borrows ideas from object-oriented programming, namely by employing subtyping and nested composition (which is closely related to inheritance). This mix of ideas, together with some new ideas, results in a language that supports highly modular programs in a natural way.

Future work. As CP is a prototype design for Compositional Programming, there is still a lot of room for making it more expressive and practical. Some possible directions for future work include the addition of *recursive types* and *type constructors*, *mutable state* for modeling imperative objects, and improvements on *type inference*.

In our view, the addition of recursive types is most pressing as there are many use cases for such signatures. For example, with recursive types, we can model binary methods [Bruce et al. 1996], or operations that return the same type that is being processed. For example, with recursive types, we should be able to model the double operation described by Zenger and Odersky [2005]. This operation doubles the literals in an arithmetic expression, where each constructor implements the following interface:

```
type Dbl = mu Exp. { double : Exp };
```

`Exp` is captured as a recursive type. On the other hand, supporting type constructors allows us to model, for example, the compositional interface of streams adapted from Biboudis et al. [2015]'s work:

```
type StreamSig<F : Type -> Type> = {
  Source : forall T. Array T -> F T;
  Map : forall T R. (T -> R) -> F T -> F R;
  FlatMap : forall T R. (T -> F R) -> F T -> F R;
  Filter : (T -> Bool) -> F T -> F T;
};
```

where the sort `F` is a type constructor (i.e. a function on types). Extending CP with recursive types and type constructors is non-trivial. The first step is to study how recursive types and type constructors interact with disjoint intersection types and other features of CP.

Although the programming style of CP is functional, a natural question is whether the ideas of Compositional Programming can be adapted into a programming model with imperative objects. There are several challenges here. One of them is to see how mutable state can be integrated into calculi with disjoint intersection types and a merge operator. A starting point in this direction is the work by Blaauwbroek [2017], which studies the addition of mutable references into calculus with intersection types and a merge operator. Another general challenge is that, once imperative objects are supported, we must face the issues of multiple inheritance in the presence of mutable state, which is a well-known source of problems.

Finally, CP has some support for type inference, such as inferring the constructor parameters from the **`implements`** clause. However, this support is rather limited. In particular, uses of polymorphic definitions must explicitly pass all type arguments. It will be interesting to investigate *local type inference* [Pierce and Turner 2000] to infer some of those arguments and improve the convenience of using polymorphic definitions. A more ambitious direction would be looking into MLsub [Dolan and Mycroft 2017] and see whether it would be possible to adapt or extend MLsub type inference to CP. The work by van den Berg [2020] is a starting point in this direction.

A FULL TYPE SYSTEM

$\boxed{\Delta, \Sigma \vdash A \Rightarrow B}$

$$\textsc{E-top}\ \ \Delta, \Sigma \vdash \top \Rightarrow \top \qquad \textsc{E-bot}\ \ \Delta, \Sigma \vdash \bot \Rightarrow \bot \qquad \textsc{E-int}\ \ \Delta, \Sigma \vdash \mathbf{Int} \Rightarrow \mathbf{Int} \qquad \textsc{E-tvar}\ \ \frac{\alpha * A \in \Delta}{\Delta, \Sigma \vdash \alpha \Rightarrow \alpha}$$

$$\textsc{E-sig}\ \ \frac{X\langle \overline{\alpha}, \overline{\beta} \rangle \mapsto C \in \Delta \qquad \overline{\Sigma \vdash S \Rightarrow \langle A, B \rangle}}{\Delta, \Sigma \vdash X\langle \overline{S} \rangle \Rightarrow [\overline{A/\alpha}, \overline{B/\beta}]C} \qquad \textsc{E-arr}\ \ \frac{\Delta, \Sigma \vdash A \Rightarrow A_1 \qquad \Delta, \Sigma \vdash B \Rightarrow B_1}{\Delta, \Sigma \vdash A \to B \Rightarrow A_1 \to B_1}$$

$$\textsc{E-and}\ \ \frac{\Delta, \Sigma \vdash A \Rightarrow A_1 \qquad \Delta, \Sigma \vdash B \Rightarrow B_1}{\Delta, \Sigma \vdash A \,\&\, B \Rightarrow A_1 \,\&\, B_1} \qquad \textsc{E-trait}\ \ \frac{\Delta, \Sigma \vdash A \Rightarrow A_1 \qquad \Delta, \Sigma \vdash B \Rightarrow B_1}{\Delta, \Sigma \vdash \mathbf{Trait}[A, B] \Rightarrow \mathbf{Trait}[A_1, B_1]}$$

$$\textsc{E-rcd}\ \ \frac{\Delta, \Sigma \vdash A \Rightarrow B}{\Delta, \Sigma \vdash \{\ell : A\} \Rightarrow \{\ell : B\}} \qquad \textsc{E-all}\ \ \frac{\Delta, \Sigma \vdash A \Rightarrow A_1 \qquad \Delta, \alpha * A_1 \vdash B \Rightarrow B_1}{\Delta, \Sigma \vdash \forall(\alpha * A).B \Rightarrow \forall(\alpha * A_1).B_1}$$

$\boxed{\Sigma \vdash S \Rightarrow \langle A, B \rangle}$

$$\textsc{E-sort1Sort}\ \ \frac{\alpha \mapsto \beta \in \Sigma}{\Sigma \vdash \alpha \Rightarrow \langle \alpha, \beta \rangle} \qquad \textsc{E-sort1}\ \ \Sigma \vdash A \Rightarrow \langle A, A \rangle \qquad \textsc{E-sort2}\ \ \Sigma \vdash A \,\%\, B \Rightarrow \langle A \,\&\, B, B \rangle$$

$\boxed{\Sigma \vdash_p^c A \Rightarrow B}$

$$\textsc{TR-top}\ \ \Sigma \vdash_p^c \top \Rightarrow \top \qquad \textsc{TR-bot}\ \ \Sigma \vdash_p^c \bot \Rightarrow \bot \qquad \textsc{TR-int}\ \ \Sigma \vdash_p^c \mathbf{Int} \Rightarrow \mathbf{Int} \qquad \textsc{TR-positive}\ \ \frac{\alpha \mapsto \beta \in \Sigma}{\Sigma \vdash_+^{\text{false}} \alpha \Rightarrow \beta}$$

$$\textsc{TR-ctrPositive}\ \ \frac{\alpha \mapsto \beta \in \Sigma}{\Sigma \vdash_+^{\text{true}} \alpha \Rightarrow \mathbf{Trait}[\alpha, \beta]} \qquad \textsc{TR-tvar}\ \ \Sigma \vdash_p^c \alpha \Rightarrow \alpha \qquad \textsc{TR-rcd}\ \ \frac{\Sigma \vdash_p^{\textsc{isCapitalized}(\ell)} A \Rightarrow B}{\Sigma \vdash_p^c \{\ell : A\} \Rightarrow \{\ell : B\}}$$

$$\textsc{TR-arr}\ \ \frac{\Sigma \vdash_{\text{flip}(p)}^c A \Rightarrow A_1 \qquad \Sigma \vdash_p^c B \Rightarrow B_1}{\Sigma \vdash_p^c A \to B \Rightarrow A_1 \to B_1} \qquad \textsc{TR-trait}\ \ \frac{\Sigma \vdash_{\text{flip}(p)}^c A \Rightarrow A_1 \qquad \Sigma \vdash_p^c B \Rightarrow B_1}{\Sigma \vdash_p^c \mathbf{Trait}[A, B] \Rightarrow \mathbf{Trait}[A_1, B_1]}$$

$$\textsc{TR-and}\ \ \frac{\Sigma \vdash_p^c A \Rightarrow A_1 \qquad \Sigma \vdash_p^c B \Rightarrow B_1}{\Sigma \vdash_p^c A \,\&\, B \Rightarrow A_1 \,\&\, B_1} \qquad \textsc{TR-all}\ \ \frac{\Sigma \vdash_p^c A \Rightarrow A_1 \qquad \Sigma \backslash \alpha \vdash_p^c B \Rightarrow B_1}{\Sigma \vdash_p^c \forall(\alpha * A).B \Rightarrow \forall(\alpha * A_1).B_1}$$

$\boxed{A <: B}$

S-REFL
$$A <: A$$

S-TRANS
$$\frac{A <: B \qquad B <: C}{A <: C}$$

S-TOPLIKE
$$A <: \rceil B \lceil$$

S-BOT
$$\bot <: A$$

S-RCD
$$\frac{A <: B}{\{\ell : A\} <: \{\ell : B\}}$$

S-ANDL
$$A \,\&\, B <: A$$

S-ANDR
$$A \,\&\, B <: B$$

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

S-AND
$$\frac{A <: B \qquad A <: C}{A <: B \,\&\, C}$$

S-FORALL
$$\frac{B_1 <: B_2 \qquad A_2 <: A_1}{\forall(\alpha * A_1).B_1 <: \forall(\alpha * A_2).B_2}$$

S-TRAIT
$$\frac{A_2 <: A_1 \qquad B_1 <: B_2}{\mathbf{Trait}[A_1, B_1] <: \mathbf{Trait}[A_2, B_2]}$$

S-DISTARR
$$(A \to B) \,\&\, (A \to C) <: A \to B \,\&\, C$$

S-DISTTRAIT
$$\mathbf{Trait}[A, B] \,\&\, \mathbf{Trait}[A, C] <: \mathbf{Trait}[A, B \,\&\, C]$$

S-DISTRCD
$$\{\ell : A\} \,\&\, \{\ell : B\} <: \{\ell : A \,\&\, B\}$$

S-DISTALL
$$\forall(\alpha * A).B \,\&\, \forall(\alpha * A).C <: \forall(\alpha * A).B \,\&\, C$$

$\boxed{\rceil A \lceil}$

TL-TOP
$$\rceil \top \lceil$$

TL-AND
$$\frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil}$$

TL-ARR
$$\frac{\rceil B \lceil}{\rceil A \to B \lceil}$$

TL-RCD
$$\frac{\rceil A \lceil}{\rceil \{\ell : A\} \lceil}$$

TL-ALL
$$\frac{\rceil B \lceil}{\rceil \forall(\alpha * A).B \lceil}$$

TL-TRAIT
$$\frac{\rceil B \lceil}{\rceil \mathbf{Trait}[A, B] \lceil}$$

$\boxed{\Delta \vdash A * B}$

D-TOPL
$$\frac{\rceil A \lceil}{\Delta \vdash A * B}$$

D-TOPR
$$\frac{\rceil B \lceil}{\Delta \vdash A * B}$$

D-ARR
$$\frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2}$$

D-ANDL
$$\frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B}$$

D-ANDR
$$\frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2}$$

D-RCDEQ
$$\frac{\Delta \vdash A * B}{\Delta \vdash \{\ell : A\} * \{\ell : B\}}$$

D-RCDNEQ
$$\frac{\ell_1 \neq \ell_2}{\Delta \vdash \{\ell_1 : A\} * \{\ell_2 : B\}}$$

D-TVARL
$$\frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B}$$

D-TVARR
$$\frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha}$$

D-FORALL
$$\frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(\alpha * A_1).B_1 * \forall(\alpha * A_2).B_2}$$

D-TRAIT
$$\frac{\Delta \vdash B_1 * B_2}{\Delta \vdash \mathbf{Trait}[A_1, B_1] * \mathbf{Trait}[A_2, B_2]}$$

D-TRAITARR
$$\frac{\Delta \vdash B_1 * B_2}{\Delta \vdash \mathbf{Trait}[A_1, B_1] * A_2 \to B_2}$$

D-ARRTRAIT
$$\frac{\Delta \vdash B_1 * B_2}{\Delta \vdash A_1 \to B_1 * \mathbf{Trait}[A_2, B_2]}$$

D-AX
$$\frac{A *_{ax} B}{\Delta \vdash A * B}$$

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

$$\textsc{Dax-intArr}\ \mathbf{Int} *_{ax} A_1 \to A_2 \qquad \textsc{Dax-intRcd}\ \mathbf{Int} *_{ax} \{\ell : A\} \qquad \textsc{Dax-intAll}\ \mathbf{Int} *_{ax} \forall(\alpha * B_1).B_2 \qquad \textsc{Dax-intTrait}\ \mathbf{Int} *_{ax} \mathbf{Trait}[A, B]$$

$$\textsc{Dax-arrAll}\ A_1 \to A_2 *_{ax} \forall(\alpha * B_1).B_2 \qquad \textsc{Dax-arrRcd}\ A_1 \to A_2 *_{ax} \{\ell : B\} \qquad \textsc{Dax-rcdTrait}\ \{\ell : C\} *_{ax} \mathbf{Trait}[A, B]$$

$$\textsc{Dax-allRcd}\ \forall(\alpha * A_1).A_2 *_{ax} \{\ell : B\} \qquad \textsc{Dax-allTrait}\ \forall(\alpha * C_1).C_2 *_{ax} \mathbf{Trait}[A, B] \qquad \textsc{Dax-sym}\ \frac{B *_{ax} A}{A *_{ax} B}$$

$\boxed{\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e}$

$$\textsc{T-tyDecl}\ \frac{\Delta; \overline{\alpha \mapsto \beta} \vdash B \Rightarrow B_1 \qquad \text{fresh } \overline{\beta} \qquad \overline{\alpha \mapsto \beta} \vdash^{\text{false}}_{+} B_1 \Rightarrow B_2 \qquad \Delta; \overline{\alpha \mapsto \beta} \vdash A \Rightarrow A_1 \qquad \Delta, X\langle\overline{\alpha}, \overline{\beta}\rangle \mapsto A_1 \,\&\, B_2; \Gamma \vdash P \Rightarrow C \rightsquigarrow e}{\Delta; \Gamma \vdash \mathbf{type}\ X\langle\overline{\alpha}\rangle\ \mathbf{extends}\ A = B; P \Rightarrow C \rightsquigarrow e}$$

$$\textsc{T-tmDecl}\ \frac{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e_1 \qquad \Delta; \Gamma, x : A \vdash P \Rightarrow B \rightsquigarrow e_2}{\Delta; \Gamma \vdash x = E; P \Rightarrow B \rightsquigarrow \mathbf{let}\ x : |A| = e_1\ \mathbf{in}\ e_2}$$

$\boxed{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e}$

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

$$\textsc{T-app}\ \frac{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta; \Gamma \vdash E_1\ E_2 \Rightarrow A_2 \rightsquigarrow e_1\ e_2} \qquad \textsc{T-anno}\ \frac{\Delta; \bullet \vdash A \Rightarrow B \qquad \Delta; \Gamma \vdash E \Leftarrow B \rightsquigarrow e}{\Delta; \Gamma \vdash E : A \Rightarrow B \rightsquigarrow e : |B|}$$

$$\textsc{T-rcd}\ \frac{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Delta; \Gamma \vdash \{\ell = E\} \Rightarrow \{\ell : A\} \rightsquigarrow e} \qquad \textsc{T-proj}\ \frac{\Delta; \Gamma \vdash E \Rightarrow \{\ell : A\} \rightsquigarrow e}{\Delta; \Gamma \vdash E.\ell \Rightarrow A \rightsquigarrow e}$$

$$\textsc{T-tabs}\ \frac{\Delta; \bullet \vdash A \Rightarrow A_1 \qquad \Delta, \alpha * A_1; \Gamma \vdash E \Rightarrow B \rightsquigarrow e}{\Delta; \Gamma \vdash \Lambda(\alpha * A).E \Rightarrow \forall(\alpha * |A_1|).B \rightsquigarrow e}$$

$$\textsc{T-tapp}\ \frac{\Delta; \bullet \vdash A \Rightarrow A_1 \qquad \Delta; \Gamma \vdash E \Rightarrow \forall(\alpha * B).C \rightsquigarrow e \qquad \Delta \vdash A_1 * B}{\Delta; \Gamma \vdash E\ @A \Rightarrow [A_1/\alpha]C \rightsquigarrow e\ |A_1|}$$

$$\textsc{T-let}\ \frac{\Delta;\bullet \vdash A \Rightarrow A_1 \qquad \Delta;\Gamma, x : A_1 \vdash E_1 \Leftarrow A_1 \rightsquigarrow e_1 \qquad \Delta;\Gamma, x : A_1 \vdash E_2 \Rightarrow B \rightsquigarrow e_2}{\Delta;\Gamma \vdash \textbf{let}\ x : A = E_1\ \textbf{in}\ E_2 \Rightarrow B \rightsquigarrow \textbf{let}\ x : |A_1| = e_1\ \textbf{in}\ e_2}$$

$$\textsc{T-mergeTrait}\ \frac{\Delta;\Gamma \vdash E_1 \Rightarrow \textbf{Trait}[A_1, B_1] \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Rightarrow \textbf{Trait}[A_2, B_2] \rightsquigarrow e_2 \qquad \Delta \vdash B_1 * B_2}{\Delta;\Gamma \vdash E_1\ ,,\ E_2 \Rightarrow \textbf{Trait}[A_1 \,\&\, A_2, B_1 \,\&\, B_2] \rightsquigarrow \lambda(\textbf{self} : |A_1 \,\&\, A_2|).\ e_1\ \textbf{self}\ ,,\ e_2\ \textbf{self}}$$

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

$$\textsc{T-new}\ \frac{\Delta;\Gamma \vdash E \Rightarrow \textbf{Trait}[A, B] \rightsquigarrow e \qquad B <: A}{\Delta;\Gamma \vdash \textbf{new}\ E \Rightarrow B \rightsquigarrow \textbf{let self} : |B| = e\ \textbf{self in self}}$$

$$\textsc{T-open}\ \frac{\Delta;\Gamma \vdash E_1 \Rightarrow \overline{\{\ell_i : A_i\}} \rightsquigarrow e_1 \qquad \Delta;\Gamma, \overline{\ell_i : A_i} \vdash E_2 \Rightarrow e_2 \rightsquigarrow B \qquad \text{fresh } x}{\Delta;\Gamma \vdash \textbf{open}\ E_1\ \textbf{in}\ E_2 \Rightarrow B \rightsquigarrow \textbf{let}\ x = e_1\ \textbf{in}\ \overline{\textbf{let}\ \ell_i : |A| = x.\ell_i}\ \textbf{in}\ e_2}$$

$$\textsc{T-trait}\ \frac{\begin{array}{c}\Delta;\bullet \vdash A \Rightarrow A_1 \qquad \Delta;\bullet \vdash B \Rightarrow B_1 \qquad \Delta;\Gamma, \textbf{self} : A_1 \vdash E_1 \Rightarrow \textbf{Trait}[A_2, B_2] \rightsquigarrow e_1 \\ A_1 <: A_2 \qquad \Delta;\Gamma, \textbf{self} : A_1, \textbf{super} : B_2 \vdash E_2 \Rightarrow C \rightsquigarrow e_2 \qquad C * B_2 \qquad C \,\&\, B_2 <: B_1\end{array}}{\begin{array}{c}\Delta;\Gamma \vdash \textbf{trait}[\textbf{self} : A]\ \textbf{implements}\ B\ \textbf{inherits}\ E_1\ \texttt{=>}\ E_2 \Rightarrow \textbf{Trait}[A_1, C \,\&\, B_2] \\ \rightsquigarrow \lambda(\textbf{self} : |A_1|).\ \textbf{let super} = e_1\ \textbf{self in}\ e_2\ ,,\ \textbf{super}\end{array}}$$

$$\textsc{T-forward}\ \frac{\Delta;\Gamma \vdash E_1 \Rightarrow \textbf{Trait}[A, B] \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Leftarrow A \rightsquigarrow e_2}{\Delta;\Gamma \vdash E_1\hat{}E_2 \Rightarrow B \rightsquigarrow e_1\ e_2}$$

$\boxed{\Delta;\Gamma \vdash E \Leftarrow A \rightsquigarrow e}$

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

$$\begin{aligned} |\textbf{Trait}[A, B]| &= |A| \to |B| \\ |A \to B| &= |A| \to |B| \\ |A \,\&\, B| &= |A| \,\&\, |B| \\ |\{\ell : A\}| &= \{\ell : |A|\} \\ |\forall(\alpha * A).B| &= \forall(\alpha * |A|).|B| \\ |A| &= A \end{aligned}$$

B METATHEORY

Lemma B.1 (Well-formedness preservation). *If* $\Delta, \Sigma \vdash A \Rightarrow B$ *then* $|\Delta| \vdash |B|$.

Proof. By simple induction on the derivation of the judgment. □

Lemma B.2 (Disjointness axiom preservation). *If $A *_{ax} B$ then $|A| *_{ax} |B|$.*

Proof. Note that $|\mathbf{Trait}[A, B]| = |A| \to |B|$; the rest are immediate. □

Lemma B.3 (Subtyping preservation). *If $A <: B$ then $|A| <: |B|$.*

Proof. Most of them are just F_i^+ subtyping. We only show the rule S-trait

$$\frac{\text{S-TRAIT}\quad A_2 <: A_1 \qquad B_1 <: B_2}{\mathbf{Trait}[A_1, B_1] <: \mathbf{Trait}[A_2, B_2]}$$

$\lvert A_2\rvert <: \lvert A_1\rvert$	By i.h.
$\lvert B_1\rvert <: \lvert B_2\rvert$	By i.h.
$\lvert A_1\rvert \to \lvert B_1\rvert <: \lvert A_2\rvert \to \lvert B_2\rvert$	By TS-arr

□

Lemma B.4 (Disjointness preservation). *If $\Delta \vdash A * B$ then $|\Delta| \vdash |A| * |B|$.*

Proof. By induction on the derivation of the judgment.

- D-topL, D-topR, and D-rcdNeq are immediate.
- $$\frac{\text{D-TVARL}\quad (\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B}$$

$\lvert A\rvert <: \lvert B\rvert$	By Lemma B.3
$a * A \in \Delta$	Given
$a * \lvert A\rvert \in \lvert\Delta\rvert$	Above
$\lvert\Delta\rvert \vdash \alpha * \lvert B\rvert$	By TD-tvarL

- $$\frac{\text{D-TVARR}\quad (\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha}$$

$\lvert A\rvert <: \lvert B\rvert$	By Lemma B.3
$a * A \in \Delta$	Given
$a * \lvert A\rvert \in \lvert\Delta\rvert$	Above
$\lvert\Delta\rvert \vdash \lvert B\rvert * \alpha$	By TD-tvarR

- $$\frac{\text{D-FORALL}\quad \Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(\alpha * A_1).B_1 * \forall(\alpha * A_2).B_2}$$

$\lvert\Delta\rvert, \alpha * \lvert A_1\rvert \,\&\, \lvert A_2\rvert \vdash \lvert B_1\rvert * \lvert B_2\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \forall(\alpha * \lvert A_1\rvert).B_1 * \forall(\alpha * \lvert A_2\rvert).\lvert B_2\rvert$	By TD-forall

- $$\frac{\Delta \vdash A * B}{\Delta \vdash \{\ell : A\} * \{\ell : B\}}\ \text{D-RCDEQ}$$

$|\Delta| \vdash |A| * |B|$ By i.h.
$|\Delta| \vdash \{\ell : |A|\} * \{\ell : |B|\}$ By TD-RCDEQ

- $$\frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2}\ \text{D-ARR}$$

$|\Delta| \vdash |A_2| * |B_2|$ By i.h.
$|\Delta| \vdash |A_1| \to |A_2| * |B_1| \to |B_2|$ By TD-ARR

- $$\frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B}\ \text{D-ANDL}$$

$|\Delta| \vdash |A_1| * |B|$ By i.h.
$|\Delta| \vdash |A_2| * |B|$ By i.h.
$|\Delta| \vdash |A_1| \,\&\, |A_2| * |B|$ By TD-ANDL

- $$\frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2}\ \text{D-ANDR}$$

$|\Delta| \vdash |A| * |B_1|$ By i.h.
$|\Delta| \vdash |A| * |B_2|$ By i.h.
$|\Delta| \vdash |A| * |B_1| \,\&\, |B_2|$ By TD-ANDR

- $$\frac{\Delta \vdash B_1 * B_2}{\Delta \vdash \mathbf{Trait}[A_1, B_1] * \mathbf{Trait}[A_2, B_2]}\ \text{D-TRAIT}$$

$|\Delta| \vdash |B_1| * |B_2|$ By i.h.
$|\Delta| \vdash |A_1| \to |B_1| * |A_2| \to |B_2|$ By TD-ARR

- $$\frac{\Delta \vdash B_1 * B_2}{\Delta \vdash \mathbf{Trait}[A_1, B_1] * A_2 \to B_2}\ \text{D-TRAITARR}$$

$|\Delta| \vdash |B_1| * |B_2|$ By i.h.
$|\Delta| \vdash |A_1| \to |B_1| * |A_2| \to |B_2|$ By TD-ARR

- $$\dfrac{\Delta \vdash B_1 * B_2}{\Delta \vdash A_1 \to B_1 * \mathbf{Trait}[A_2, B_2]}\ \textsc{D-arrTrait}$$

$|\Delta| \vdash |B_1| * |B_2|$ By i.h.
$|\Delta| \vdash |A_1| \to |B_1| * |A_2| \to |B_2|$ By TD-arr

- $$\dfrac{A *_{ax} B}{\Delta \vdash A * B}\ \textsc{D-ax}$$

$|A| *_{ax} |B|$ By Lemma B.2
$|\Delta| \vdash |A| * |B|$ By TD-ax

□

Theorem B.5 (Type-safety). *We have that:*
If $\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e$ *then* $|\Delta|; |\Gamma| \vdash e \Rightarrow |A|$.
If $\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e$ *then* $|\Delta|; |\Gamma| \vdash e \Rightarrow |A|$.
If $\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e$ *then* $|\Delta|; |\Gamma| \vdash e \Leftarrow |A|$.

Proof. By induction on the typing judgment.

- $$\dfrac{\begin{array}{c}\text{fresh } \overline{\beta} \qquad \Delta; \overline{\alpha \mapsto \beta} \vdash A \Rightarrow A_1 \qquad \Delta; \overline{\alpha \mapsto \beta} \vdash B \Rightarrow B_1 \\ \overline{\alpha \mapsto \beta} \vdash_{+}^{\text{false}} B_1 \Rightarrow B_2 \qquad \Delta, X\langle \overline{\alpha}, \overline{\beta} \rangle \mapsto A_1 \,\&\, B_2; \Gamma \vdash P \Rightarrow C \rightsquigarrow e\end{array}}{\Delta; \Gamma \vdash \mathbf{type}\ X\langle \overline{\alpha} \rangle\ \mathbf{extends}\ A = B; P \Rightarrow C \rightsquigarrow e}\ \textsc{T-tyDecl}$$

$|\Delta|; |\Gamma| \vdash e \Rightarrow |C|$ By i.h.

- $$\dfrac{\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e_1 \qquad \Delta; \Gamma, x : A \vdash P \Rightarrow B \rightsquigarrow e_2}{\Delta; \Gamma \vdash x = E; P \Rightarrow B \rightsquigarrow \mathbf{let}\ x : |A| = e_1\ \mathbf{in}\ e_2}\ \textsc{T-tmDecl}$$

$|\Delta|; |\Gamma| \vdash e_1 \Rightarrow |A|$ By i.h.
$|\Delta|; |\Gamma|, x : |A| \vdash e_2 \Rightarrow |B|$ By i.h.
$|\Delta|; |\Gamma| \vdash \mathbf{let}\ x : |A| = e_1\ \mathbf{in}\ e_2 \Rightarrow |B|$ By TT-let

- T-top, T-nat, and T-var are immediate.
- $$\dfrac{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta; \Gamma \vdash E_1\ E_2 \Rightarrow A_2 \rightsquigarrow e_1\ e_2}\ \textsc{T-app}$$

$|\Delta|; |\Gamma| \vdash e_1 \Rightarrow |A_1| \to |A_2|$ By i.h.
$|\Delta|; |\Gamma| \vdash e_2 \Leftarrow |A_2|$ By i.h.
$|\Delta|; |\Gamma| \vdash e_1\ e_2 \Rightarrow |A_2|$ By TT-app

- $$\frac{\Delta; \bullet \vdash A \Rightarrow B \qquad \Delta; \Gamma \vdash E \Leftarrow B \rightsquigarrow e}{\Delta; \Gamma \vdash E : A \Rightarrow B \rightsquigarrow e : |B|}\ \textsc{T-anno}$$

$|\Delta|; |\Gamma| \vdash e \Leftarrow |B|$ By i.h.
$|\Delta|; |\Gamma| \vdash e : |B| \Rightarrow |B|$ By TT-anno

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

$|\Delta|; |\Gamma| \vdash e \Rightarrow |A|$ By i.h.
$|\Delta|; |\Gamma| \vdash \{\ell = e\} \Rightarrow \{\ell : |A|\}$ By TT-rcd

- $$\frac{\Delta; \Gamma \vdash E \Rightarrow \{\ell : A\} \rightsquigarrow e}{\Delta; \Gamma \vdash E.\ell \Rightarrow A \rightsquigarrow e}\ \textsc{T-proj}$$

$|\Delta|; |\Gamma| \vdash e \Rightarrow \{\ell : |A|\}$ By i.h.
$|\Delta|; |\Gamma| \vdash e.\ell \Rightarrow |A|$ By TT-proj

- $$\frac{\Delta; \bullet \vdash A \Rightarrow A_1 \qquad \Delta, \alpha * A_1; \Gamma \vdash E \Rightarrow B \rightsquigarrow e}{\Delta; \Gamma \vdash \Lambda(\alpha * A).E \Rightarrow \forall(\alpha * |A_1|).B \rightsquigarrow e}\ \textsc{T-tabs}$$

$|\Delta| \vdash |A_1|$ By Lemma B.1
$|\Delta|; |\Gamma|, \alpha * |A| \vdash e \Rightarrow |B|$ By i.h.
$|\Delta|; |\Gamma| \vdash \Lambda(\alpha * |A|).\, e \Rightarrow \forall(\alpha * |A|).\, |B|$ By TT-tabs

- $$\frac{\Delta; \bullet \vdash A \Rightarrow A_1 \qquad \Delta; \Gamma \vdash E \Rightarrow \forall(\alpha * B).C \rightsquigarrow e \qquad \Delta \vdash A_1 * B}{\Delta; \Gamma \vdash E\ @A \Rightarrow [A_1/\alpha]C \rightsquigarrow e\ |A_1|}\ \textsc{T-tapp}$$

$|\Delta| \vdash |A_1|$ By Lemma B.1
$|\Delta|; |\Gamma| \vdash e \Rightarrow \forall(\alpha * |B|).|C|$ By i.h.
$|\Delta|; |\Gamma| \vdash |A_1| * |B|$ By Lemma B.4
$|\Delta|; |\Gamma| \vdash e\ |A| \Rightarrow [|A_1|/\alpha]|C|$ By TT-tapp

- $$\frac{\Delta; \bullet \vdash A \Rightarrow A_1 \qquad \Delta; \Gamma, x : A_1 \vdash E_1 \Leftarrow A_1 \rightsquigarrow e_1 \qquad \Delta; \Gamma, x : A_1 \vdash E_2 \Rightarrow B \rightsquigarrow e_2}{\Delta; \Gamma \vdash \textbf{let}\ x : A = E_1\ \textbf{in}\ E_2 \Rightarrow B \rightsquigarrow \textbf{let}\ x : |A_1| = e_1\ \textbf{in}\ e_2}\ \textsc{T-let}$$

$|\Delta| \vdash |A_1|$ By Lemma B.1
$|\Delta|; |\Gamma|, x : |A_1| \vdash e_1 \Leftarrow |A|$ By i.h.
$|\Delta|; |\Gamma|, x : |A_1| \vdash e_2 \Rightarrow |B|$ By i.h.
$|\Delta|; |\Gamma| \vdash \textbf{let}\ x : |A_1| = E_1\ \textbf{in}\ E_2 \Rightarrow |B|$ By TT-let

- T-mergeTrait
$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow \mathbf{Trait}[A_1, B_1] \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Rightarrow \mathbf{Trait}[A_2, B_2] \rightsquigarrow e_2 \qquad \Delta \vdash B_1 * B_2}{\Delta;\Gamma \vdash E_1 \,,,\, E_2 \Rightarrow \mathbf{Trait}[A_1 \,\&\, A_2, B_1 \,\&\, B_2] \rightsquigarrow \lambda(\mathbf{self}{:}|A_1 \,\&\, A_2|).\, e_1\ \mathbf{self} \,,,\, e_2\ \mathbf{self}}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_1 \Rightarrow \lvert A_1 \to B_1\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_2 \Rightarrow \lvert A_2 \to B_2\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \mathbf{self} : \lvert A_1 \,\&\, A_2\rvert \vdash \mathbf{self} \Rightarrow \lvert A_1 \,\&\, A_2\rvert$	By TT-var
$\lvert A_1 \,\&\, A_2\rvert <: \lvert A_1\rvert$	By TS-andL
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash \mathbf{self} \Leftarrow \lvert A_1\rvert$	By TT-sub
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_1\ \mathbf{self} \Rightarrow \lvert B_1\rvert$	By TT-app
$\lvert A_1 \,\&\, A_2\rvert <: \lvert A_2\rvert$	By TS-andR
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash \mathbf{self} \Leftarrow \lvert A_2\rvert$	By TT-sub
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_2\ \mathbf{self} \Rightarrow \lvert B_2\rvert$	By TT-app
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_1\ \mathbf{self} \,,,\, e_2\ \mathbf{self} \Rightarrow \lvert B_1 \,\&\, B_2\rvert$	By TT-merge
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash \lambda(\mathbf{self}{:}\lvert A_1 \,\&\, A_2\rvert).\, e_1\ \mathbf{self} \,,,\, e_2\ \mathbf{self} \Rightarrow \lvert A_1 \,\&\, A_2\rvert \to \lvert B_1 \,\&\, B_2\rvert$	By TT-abs

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

$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_1 \Rightarrow \lvert A\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_2 \Rightarrow \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash \lvert A\rvert * \lvert B\rvert$	By Lemma B.4
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_1,, e_2 \Rightarrow \lvert A\rvert \,\&\, \lvert B\rvert$	By TT-merge

- T-new
$$\frac{\Delta;\Gamma \vdash E \Rightarrow \mathbf{Trait}[A, B] \rightsquigarrow e \qquad B <: A}{\Delta;\Gamma \vdash \mathbf{new}\ E \Rightarrow B \rightsquigarrow \mathbf{let\ self} : \lvert B\rvert = e\ \mathbf{self\ in\ self}}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e \Rightarrow \lvert A\rvert \to \lvert B\rvert$	By i.h.
$\lvert B\rvert <: \lvert A\rvert$	By Lemma B.3
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \mathbf{self} : \lvert B\rvert \vdash \mathbf{self} \Rightarrow \lvert B\rvert$	By TT-var
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \mathbf{self} : \lvert B\rvert \vdash \mathbf{self} \Leftarrow \lvert A\rvert$	By TT-sub
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \mathbf{self} : \lvert B\rvert \vdash e\ \mathbf{self} \Leftarrow \lvert B\rvert$	By TT-app
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash \mathbf{let\ self} : \lvert B\rvert = e\ \mathbf{self\ in\ self} \Rightarrow \lvert B\rvert$	By TT-let

- T-open
$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow \overline{\{\ell_i : A_i\}} \rightsquigarrow e_1 \qquad \Delta;\Gamma, \overline{\ell_i : A_i} \vdash E_2 \Rightarrow e_2 \rightsquigarrow B \qquad \text{fresh } x}{\Delta;\Gamma \vdash \mathbf{open}\ E_1\ \mathbf{in}\ E_2 \Rightarrow B \rightsquigarrow \mathbf{let}\ x = e_1\ \mathbf{in}\ \overline{\mathbf{let}\ \ell_i : \lvert A\rvert = x.\ell_i}\ \mathbf{in}\ e_2}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_1 \Rightarrow \lvert\overline{\{\ell_i : A_i\}}\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_1.\ell_i \Rightarrow \lvert A_i\rvert$	By TT-proj
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash \overline{\mathbf{let}\ \ell_i : \lvert A_i\rvert = e_1.\ell_i}\ \mathbf{in}\ e_2 \Rightarrow \lvert B\rvert$	By TT-let

- T-TRAIT

$$\frac{\Delta; \bullet \vdash A \Rightarrow A_1 \qquad \Delta; \bullet \vdash B \Rightarrow B_1 \qquad \Delta; \Gamma, \mathbf{self} : A_1 \vdash E_1 \Rightarrow \mathbf{Trait}[A_2, B_2] \rightsquigarrow e_1 \qquad A_1 <: A_2 \qquad \Delta; \Gamma, \mathbf{self} : A_1, \mathbf{super} : B_2 \vdash E_2 \Rightarrow C \rightsquigarrow e_2 \qquad C * B_2 \qquad C \,\&\, B_2 <: B_1}{\Delta; \Gamma \vdash \mathbf{trait}[\mathbf{self} : A]\ \mathbf{implements}\ B\ \mathbf{inherits}\ E_1\ \texttt{=>}\ E_2 \Rightarrow \mathbf{Trait}[A_1, C \,\&\, B_2] \rightsquigarrow \lambda(\mathbf{self} : |A_1|).\ \mathbf{let\ super} = e_1\ \mathbf{self\ in}\ e_2\ ,,\ \mathbf{super}}$$

$\lvert\Delta\rvert \vdash \lvert A_1\rvert$	By Lemma B.1
$\lvert\Delta\rvert \vdash \lvert B_1\rvert$	By Lemma B.1
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathbf{self} : \lvert A_1\rvert \vdash e_1 \Rightarrow \lvert A_2\rvert \to \lvert B_2\rvert$	By i.h.
$\lvert A_1\rvert <: \lvert A_2\rvert$	By Lemma B.3
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathbf{self} : \lvert A_1\rvert, \mathbf{super} : \lvert B_2\rvert \vdash e_2 \Rightarrow \lvert C\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathbf{self} : \lvert A_1\rvert \vdash \mathbf{self} \Rightarrow \lvert A_1\rvert$	By TT-VAR
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathbf{self} : \lvert A_1\rvert \vdash \mathbf{self} \Leftarrow \lvert B_1\rvert$	By TT-SUB
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathbf{self} : \lvert A_1\rvert \vdash e_1\ \mathbf{self} \Rightarrow \lvert B_2\rvert$	By TT-TAPP
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \mathbf{self} : \lvert A_1\rvert, \mathbf{super} : \lvert B_2\rvert \vdash e_2\ ,,\ \mathbf{super} \Rightarrow \lvert C\rvert \,\&\, \lvert B_2\rvert$	By TT-MERGE
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \mathbf{let\ super} = e_1\ \mathbf{self\ in}\ e_2\ ,,\ \mathbf{super} \Rightarrow \lvert C\rvert \,\&\, \lvert B_2\rvert$	By TT-LET
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \lambda(\mathbf{self} : \lvert A_1\rvert).\ \mathbf{let\ super} = e_1\ \mathbf{self\ in}\ e_2\ ,,\ \mathbf{super} \Rightarrow \lvert A_1\rvert \to (\lvert C\rvert \,\&\, \lvert B_2\rvert)$	By TT-ABS

- T-FORWARD

$$\frac{\Delta; \Gamma \vdash E_1 \Rightarrow \mathbf{Trait}[A, B] \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash E_2 \Leftarrow A \rightsquigarrow e_2}{\Delta; \Gamma \vdash E_1\hat{}E_2 \Rightarrow B \rightsquigarrow e_1\ e_2}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_1 \Rightarrow \lvert A\rvert \to \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_2 \Leftarrow \lvert A\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_1\ e_2$	By TT-APP

- T-ABS

$$\frac{\Delta; \Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Delta; \Gamma \vdash \lambda x.E \Leftarrow A \to B \rightsquigarrow \lambda x.E}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert, x : \lvert A\rvert \vdash e \Leftarrow \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \lambda(x : e).\ \Leftarrow \lvert A\rvert \to \lvert B\rvert$	By TT-ABS

- T-SUB

$$\frac{\Delta; \Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad B <: A}{\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e \Rightarrow \lvert B\rvert$	By i.h.
$\lvert B\rvert <: \lvert A\rvert$	By Lemma B.3
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e \Leftarrow \lvert B\rvert$	By TT-SUB

□

ACKNOWLEDGMENTS

We thank anonymous reviewers for their helpful comments. We also appreciate Fengmin Zhu who pointed out a minor flaw in the rules of disjointness in Appendix A after the paper was published. (The rules are fixed in this revision.)

This work has been sponsored by the Hong Kong Research Grant Council projects number 17210617 and 17209519. The first author conducted this research while at the University of Hong Kong.

REFERENCES

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint polymorphism. In *European Symposium on Programming*. Springer, 1–28.

Davide Ancona and Elena Zucca. 2002. A calculus of module systems. *Journal of functional programming* 12, 2 (2002), 91–132.

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. 2006. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*. Springer, 135–173.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment 1. *The journal of symbolic logic* 48, 4 (1983), 931–940.

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed first-class traits. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The essence of nested composition. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *European Symposium on Programming*. Springer, 381–409.

Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. 2015. Streams a la carte: Extensible pipelines with object algebras. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Lasse Blaauwbroek. 2017. *On the Interaction Between Unrestricted Union and Intersection Types and Computational Effects*. Master's thesis. Technical University Eindhoven.

Gilad Bracha. 1992. *The programming language jigsaw: mixins, modularity and multiple inheritance*. Ph.D. Dissertation. Dept. of Computer Science, University of Utah.

Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90)*.

Gilad Bracha, Peter Von Der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as objects in newspeak. In *European Conference on Object-Oriented Programming*. Springer, 405–428.

K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. 1996. On Binary Methods. *Theory and Practice of Object Systems* 1, 3 (1996). To appear.

Kim Bruce, Martin Odersky, and Philip Wadler. 1998. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming*.

R. M. Burstall, D. B. MacQueen, and D. T. Sannella. 1981. *HOPE: An experimental applicative Language*. Technical Report CSR-62-80. Computer Science Dept, Univ. of Edinburgh.

Luca Cardelli. 1994. *Extensible Records in a Pure Calculus of Subtyping*. MIT Press, Cambridge, MA, USA.

Luca Cardelli and John Mitchell. 1991. Operations on Records. *Mathematical Structures in Computer Science* 1 (1991), 3–48.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming* 19, 05 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

James Cheney and Ralf Hinze. 2002. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell (Haskell '02)* (Pittsburgh, PA, USA), Manuel M.T. Chakravarty (Ed.). ACM, New York, NY, USA, 90–104. https://doi.org/10.1145/581690.581698

Adam Chlipala. 2010. Ur: statically-typed metaprogramming with type-level record computation. *ACM SIGPLAN Notices* 45, 6 (2010), 122–133.

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: a simple virtual class calculus. In *Proceedings of the 6th international conference on Aspect-oriented software development*. 121–134.

William Cook and Jens Palsberg. 1989. A Denotational Semantics of Inheritance and Its Correctness. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89)*. ACM, New York, NY, USA, 433–443. https://doi.org/10.1145/74877.74922

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882

Dominic Duggan and Constantinos Sourelis. 1996. Mixin modules. *ACM SIGPLAN Notices* 31, 6 (1996), 262–273.

Joshua Dunfield. 2014. Elaborating Intersection and Union Types. *Journal of Functional Programming* 24, 2-3 (2014), 133–165. https://doi.org/10.1017/S0956796813000270

Erik Ernst. 1999. *gbeta-a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Ph.D. Dissertation. University of Aarhus.

Erik Ernst. 2001. Family Polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*.

Erik Ernst. 2004. The expression problem, Scandinavian style. *On Mechanisms For Specialization* (2004), 27.

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A Virtual Class Calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 171–183.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Jacques Garrigue. 2000. Code reuse through polymorphic variants. In *In Workshop on Foundations of Software Engineering*.

Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. 339–347. https://doi.org/10.1145/2628136.2628138

John V. Guttag and James J. Horning. 1978. The algebraic specification of abstract data types. *Acta informatica* 10, 1 (1978), 27–52.

William Harrison and Harold Ossher. 1993. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. 411–428.

Ralf Hinze. 2004. An Algebra of Scans. In *Mathematics of Program Construction*. 186–210. https://doi.org/10.1007/978-3-540-27764-4_11

Ralf Hinze. 2006. Generics for the Masses. *Journal of Functional Programming* 16, 4-5 (2006), 451–483. https://doi.org/10.1017/S0956796806006022

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic Embedding of Dsls. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE '08)*.

Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.26

Atsushi Igarashi and Benjamin C Pierce. 1999. Foundations for virtual types. In *European Conference on Object-Oriented Programming*. Springer, 161–185.

Atsushi Igarashi, Chieri Saito, and Mirko Viroli. 2005. Lightweight family polymorphism. In *Asian Symposium on Programming Languages and Systems*. Springer, 161–177.

Pablo Inostroza and Tijs van der Storm. 2015. Modular Interpreters for the Masses: Implicit Context Propagation Using Object Algebras. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*.

Mauro Jaskelioff. 2008. Monatron: An extensible monad transformer library. In *Symposium on Implementation and Application of Functional Languages*. Springer, 233–248.

Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. 2004. Simple dependent types: Concord. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP)*.

Uwe Kastens and William M. Waite. 1994. Modularity and reusability in attribute grammars. *Acta Informatica* 31, 7 (1994), 601–627.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European conference on object-oriented programming*. Springer, 220–242.

Jorgen Lindskov Knudsen, Boris Magnusson, Mats Lofgren, and Ole L Madsen. 1994. *Object Oriented Software Development Environments: The Mjolner Approach*. Prentice-Hall, Inc.

Donald E. Knuth. 1968. Semantics of Context-Free Languages. *Math. Sys. Theory* 2, 2 (1968), 127–145.

Donald E. Knuth. 1990. The Genesis of Attribute Grammars. In *WAGA*. 1–12.

Giovanni Lagorio, Marco Servetto, and Elena Zucca. 2009. Featherweight jigsaw: A minimal core calculus for modular composition of classes. In *European Conference on Object-Oriented Programming*. Springer, 244–268.

Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A theory of tagged objects. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Daan Leijen. 2005. Extensible records with scoped labels. *Trends in Functional Programming* 6 (2005), 179–194.

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

David MacQueen. 1984. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. 198–207.

Ole Lehrmann Madsen and Birger Moller-Pedersen. 1989. Virtual classes: A powerful mechanism in object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*. 397–406.

Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. 1993. *Object-oriented programming in the BETA programming language*. Addison-Wesley.

Bertrand Meyer. 1988. *Object-Oriented Software Construction*. Prentice Hall.

Keiko Nakata and Jacques Garrigue. 2006. Recursive modules for programming. *ACM SIGPLAN Notices* 41, 9 (2006), 74–86.

Nathaniel Nystrom, Stephen Chong, and Andrew C Myers. 2004. Scalable extensibility via nested inheritance. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 99–115.

Nathaniel Nystrom, Xin Qi, and Andrew C Myers. 2006. J& nested intersection for scalable software composition. *ACM SIGPLAN Notices* 41, 10 (2006), 21–36.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language*. Technical Report.

Martin Odersky and Matthias Zenger. 2005. Scalable Component Abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*.

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP '12)*. https://doi.org/10.1007/978-3-642-31057-7_2

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. 2006a. Extensible and modular generics for the masses. *Trends in Functional Programming* 7 (2006), 199–216.

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. 2006b. Extensible and Modular Generics for the Masses. In *Trends in Functional Programming*. 199–216.

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 364–377.

Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. 2013. Feature-Oriented Programming with Object Algebras. In *Proceedings of the 27th European Conference on Object-Oriented Programming*. https://doi.org/10.1007/978-3-642-39038-8_2

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 44.

Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

Erik Poll. 1997. System F with Width-Subtyping and Record Updating. In *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software (TACS 1997)*. Springer-Verlag, Berlin, Heidelberg.

Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*. Springer, 419–443.

Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2014. From Object Algebras to Attribute Grammars. In *Proceedings of the 2014 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*.

Claudio V Russo. 2000. First-class structures for Standard ML. In *European Symposium on Programming*. Springer, 336–350.

Claudio V Russo. 2001. Recursive structures for Standard ML. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 50–61.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*. Springer, 248–274.

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

M. Sulzmann, M. M. T. Chakravarty, S. L. Peyton-Jones, and K. Donnelly. 2007. System F with type equality coercions. In *TLDI*.

Wouter Swierstra. 2008. Data Types à la Carte. *Journal of Functional Programming* 18, 04 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Asumu Takikawa, T Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual typing for first-class classes. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 793–810.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*. IEEE, 107–119.

Mads Torgersen. 2004. The Expression Problem Revisited. In *European Conference on Object-Oriented Programming (ECOOP)*.

Birthe van den Berg. 2020. ICFP: G: Type Inference for Disjoint Intersection Types.

Philip Wadler. 1992. The essence of functional programming. *19th POPL* (Jan. 1992), 1–14.

Philip Wadler. 1998. The Expression Problem. (Nov. 1998). Note to Java Genericity mailing list.

Yanlin Wang and Bruno C. d. S. Oliveira. 2016. The Expression Problem, Trivially!. In *Proceedings of the 15th International Conference on Modularity (MODULARITY 2016)*. 37–41. https://doi.org/10.1145/2889443.2889448

Mathhias Zenger and Martin Odersky. 2005. Independently Extensible Solutions to the Expression Problem. In *Foundations of Object-Oriented Languages (FOOL)*.

Weixin Zhang and Bruno C. d. S. Oliveira. 2017. EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse. In *31st European Conference on Object-Oriented Programming*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.29

Weixin Zhang and Bruno C. d. S. Oliveira. 2019. Shallow EDSLs and Object-Oriented Programming: Beyond Simple Compositionality. *The Art, Science, and Engineering of Programming* 3, 3 (2019), 1–25.

Yizhou Zhang and Andrew C Myers. 2017. Familia: unifying interfaces, type classes, and family polymorphism. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–31.