Type-Safe Compilation of Dynamic Inheritance via Merging

YAOZHU SUN, The University of Hong Kong, China and National Institute of Informatics, Japan
XUEJING HUANG, The University of Hong Kong, China and IRIF, Université Paris Cité, France
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

Inheritance is a key concept in many programming languages. Dynamically typed languages, such as JavaScript, often support powerful forms of dynamic inheritance. However, dynamic inheritance poses significant challenges for static typing. Most statically typed languages only provide static inheritance to achieve type safety at the cost of flexibility.

This paper presents a compiler for the CP language, which is a *statically* typed language that supports *dynamic* inheritance via a merge operator and also has an expressive form of parametric polymorphism. The merge operator enables a form of multiple inheritance and first-class classes, as well as virtual classes and family polymorphism. With these features, CP allows the development of highly modular and loosely coupled components. However, the efficient compilation of CP code is non-trivial, especially if separate compilation is desired. In particular, subtyping in CP is coercive for type safety, which poses significant challenges in obtaining an efficient compilation scheme. We show how CP is compilable to languages supporting extensible records or similar data structures, where record labels are generated from types for efficient lookup on merges. The main ideas of the compilation scheme are formalized in Coq and proven to be type-safe. The concrete implementation of the CP compiler targets JavaScript, where records are modeled as JavaScript objects. We conduct an empirical evaluation with various benchmarks and evaluate the impact of several CP-specific optimizations. With our optimizations, CP can be orders of magnitude faster than with a naive compilation scheme for merges, obtaining performance on par with class-based JavaScript programs.

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

Additional Key Words and Phrases: Compositional Programming, Separate Compilation

1 Introduction

Many programming language constructs are first-class. First-class functions are a key construct of functional programming. Similarly, objects are first-class in object-oriented programming (OOP). First-class constructs enable the corresponding values to be abstracted by variables, passed as arguments, or returned by functions or methods.

While classes are pervasive in most OOP languages, first-class classes are *much less* studied, and they are *rarely* supported in mainstream statically typed OOP languages. Languages such as Java, C#, and Swift, just to name a few, do not support first-class classes. In these languages, no variables can abstract over classes, and thus a class cannot pick which class to inherit from at run time. Nevertheless, some dynamically typed languages treat classes as first-class constructs and

Authors' Contact Information: Yaozhu Sun, The University of Hong Kong, China and National Institute of Informatics, Japan, yzsun@cs.hku.hk; Xuejing Huang, The University of Hong Kong, China and IRIF, Université Paris Cité, France, xjhuang@cs.hku.hk; Bruno C. d. S. Oliveira, The University of Hong Kong, China, bruno@cs.hku.hk.

ACM 1558-4593/2025/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
class A {                      class B extends A {
  m() { return 48 }              m() { return "Hi" }
}                              }
```

Fig. 1. JavaScript allows unconstrained overriding, whereas TypeScript's type system attempts to prevent type-unsafe overriding and statically rejects the above example.

allow dynamic inheritance. Taking JavaScript[1] as an example, a base class can be passed as an argument, and the inheritance hierarchy is determined at run time, after the application of the function happens:

```
function Mixin(Base) {
  return class extends Base {
    greet() { alert('Hello, world!') }
  };
}
```

First-class classes offer powerful and flexible abstraction mechanisms for programmers. For instance, *mixins* [Bracha and Cook 1990], which are class-like abstractions that can be mixed into other classes to add new features, are encodable via first-class classes and dynamic inheritance. In our example, the `Mixin` function creates a class that inherits from `Base` and adds a `greet` method. At run time, we can apply `Mixin` to different base classes that need `greet`. Dynamic inheritance rejects the common assumption that inheritance hierarchies are fixed at compile time, providing a greater degree of flexibility compared to static inheritance. Furthermore, first-class classes provide natural support for *nested classes*: classes defined within another class, or even inside methods or functions as in the `Mixin` example. Nested classes can access definitions and methods in the surrounding lexical scope. In JavaScript, nested classes are supported via first-class classes. Some other OOP languages, such as Java, support nested classes without supporting first-class classes.

To ensure type-safe inheritance, an important concern is how to deal with *overriding* and, more generally, method conflicts. JavaScript deals with method conflicts by employing *implicit* overriding. That is, a method in a subclass will override a method in the superclass if the superclass contains a method with the *same name*. Otherwise, a new method is defined in the subclass if no method with the same name exists in the superclass. In JavaScript or other dynamically typed languages, overriding is completely unconstrained, allowing the overriding method to return a different type. An example is shown in Fig. 1. Such overriding is not type-safe if an object of the subclass `B` is to be used in the place where the superclass `A` is expected, since the method `m` in `A` is expected to return a number instead of a string.

Since TypeScript is a superset of JavaScript, it adopts the same implicit overriding approach. However, like most statically typed OOP languages, TypeScript places restrictions on overriding to ensure type safety. In TypeScript, overriding methods must have types compatible with the overridden ones, in order to allow for the safe use of a subclass in the place of a superclass. Another possibility is to allow subclasses not to be subtypes of the superclass [Cook et al. 1990], which is sometimes seen in structurally typed OOP languages. In this case, a subclass may not always be used in place of a superclass, and a type system can prevent the use of subclasses that are not subtypes. Nevertheless, this does not imply that overriding can be fully unconstrained, as it is still possible to have type-safety issues even when inheritance does not imply subtyping.

[1]Although object-orientation in JavaScript is originally prototype-based, newer standards (ECMAScript 6+) also support classes on top of prototypes.

First-class classes and dynamic inheritance make type-safe overriding much harder. Few statically typed languages attempt to support such features, and some of the ones that do have type-unsound designs. For instance, in addition to supporting conventional static inheritance idioms, TypeScript also supports dynamic inheritance, but its type system cannot always ensure type-safe overriding. With dynamic inheritance, the exact type of the superclass is unknown statically, so it is hard to guarantee that no method is accidentally overridden with an incompatible type at run time. We will illustrate this point using examples in TypeScript later in this paper.

Implicit overriding is not the only way to deal with method conflicts. Another possibility is to detect and prevent conflicts, *disallowing* any form of implicit overriding. For instance, the *trait* model [Ducasse et al. 2006] adopts an approach where implicit overriding is disallowed. With traits overriding is still possible, but it must be *explicitly* triggered by the programmer, instead of being implicitly done by the compiler. For instance, when composing two traits with conflicts, the composition will be *rejected.* To resolve conflicts, a programmer can, for example, decide to take one of the implementations for the method, or provide a new method implementation instead.

Yet another possibility to deal with conflicts is what we call *merging* in this paper. Merging is not a new idea and has been used to a certain degree in existing programming language designs. For instance, merging is central in programming language designs with *virtual classes* [Clarke et al. 2007; Ernst et al. 2006; Madsen and Møller-Pedersen 1989] and *family polymorphism* [Ernst 2001; Saito et al. 2008; Zhang and Myers 2017]. Virtual classes are a form of nested classes. However, the main feature of virtual classes is that, when a virtual class conflicts with another virtual class with the same name, the old class is *not overridden.* Instead, the behaviors of the two classes are *merged*: the new class will contain all the methods of the old class as well as the new methods. So, unlike overriding, merging does not replace existing behaviors. Instead, it preserves existing behaviors and adds some new ones.[2]

The idea of merging can be extended to deal with conventional methods as well. For example, in a language that adopts merging, code similar to that in Fig. 1 can be accepted. Class `B` would contain *two* versions of the method `m`: one returning a number and the other returning a string. In other words, merging would act as a kind of *overloading* in this case, enabling two methods with the same name but different types to coexist in the same class. Invocations of `m` could be disambiguated by the surrounding context or, if needed, by the programmer. Of course, in the merging model, the combination of two methods with the same name and *related types* would still be problematic, as it would not be clear how to choose and disambiguate between the two method implementations.

A solution to this problem is to adopt a trait-like model with merging. This model has been adopted by the CP language [Zhang et al. 2021] and is the focus of this paper. In a trait-like model with merging, method conflicts are still forbidden, but methods with the same name and *disjoint types* (i.e. the types are unrelated) do not create a conflict. In other words, if we compose two traits, each having a method with the same name and related types, then we get an *error* due to a method conflict. However, if the methods have the same name but disjoint types, then the composition is accepted, and the resulting object will retain the two method implementations. By allowing merging in the disjoint case, we can express the forms of composition that are required for virtual classes. In such cases, virtual classes are modeled as fields, and two virtual classes with the same name but disjoint interfaces (i.e. the types of the virtual class methods) can be merged.

Such a model offers important advantages over designs that adopt implicit overriding instead. A first advantage is that we can obtain flexible, powerful, and type-safe inheritance models and avoid

[2]Strictly speaking, most designs with virtual classes will combine merging with overriding, in the case that the two virtual classes have conflicting methods. In our discussion, when we describe merging, we assume that the sets of methods in the two virtual classes are disjoint and, consequently, no overriding takes place.

many of the restrictions imposed by languages with static inheritance. With merging it is possible to have a model of inheritance that allows *dynamic inheritance* and forms of *multiple inheritance* and *family polymorphism* all at once. We are aware of some designs with dynamic inheritance and first-class classes [Lee et al. 2015; Takikawa et al. 2012], but without family polymorphism. We are also aware of some designs with both multiple inheritance and family polymorphism [Aracic et al. 2006; Clarke et al. 2007; Nystrom et al. 2006], but without dynamic inheritance. Except for the CP language, which is our focus in this paper, the only statically typed language we know that supports all three features is `gbeta` [Ernst 2000]. However, `gbeta` cannot statically guarantee that every use of dynamic inheritance is type-safe, although Ernst [2002] proves a subset of use cases to be type-safe. Moreover, separate compilation can only be supported with an inefficient linear search through super-mixins for inherited attributes. To the best of our knowledge, CP is the only language that supports all three features in a completely type-safe manner, without compromising on modular type checking or separate compilation. We believe that this absence in the design space is because it is hard to have flexible and type-safe designs that support all these features at once.

Because our design is based on the trait model, we also inherit its advantages. In particular, since merging extends behavior rather than replace behavior, it is less prone to problems such as the *fragile base class problem* [Mikhajlov and Sekerinski 1998]. As we shall see, in the presence of dynamic inheritance, designs based on implicit overriding, such as TypeScript, exacerbate the fragile base class problem: not only can overriding break invariants of the superclass, but it can also break type safety! Designs based on a trait-model with merging preserve the behavior of inherited classes and avoid the issues due to (implicit) overriding.

CP, short for *compositional programming* [Zhang et al. 2021], is a statically typed language that supports dynamic inheritance and adopts a trait model with merging. CP-flavored traits are first-class constructs [Bi and Oliveira 2018]. Thus, dynamic inheritance between traits is possible. Moreover, trait inheritance in CP is built upon nested composition [Bi et al. 2018], which enables a form of family polymorphism and virtual classes. The foundations of CP are well studied. Several statically typed calculi based on disjoint intersection types [Oliveira et al. 2016] and a merge operator [Dunfield 2014; Reynolds 1997] have been developed with small-step operational semantics [Fan et al. 2022; Huang et al. 2021] or elaboration semantics [Alpuim et al. 2017; Bi et al. 2018, 2019; Oliveira et al. 2016]. The current implementation of CP by Sun et al. [2022] employs an interpreter that is built upon the operational semantics studied in past work. Unfortunately, the interpreter-based implementation is simple but inefficient.

This paper presents a CP compiler, supporting modular type checking and separate compilation. Our primary source of inspiration comes from the elaboration approach by Dunfield [2014], where intersection types and merges are compiled into product types and pairs. However, her work lacked the distributive subtyping rules that are needed to achieve family polymorphism. More importantly, her focus was on proving type safety, and she did not investigate ways to optimize the coercive form of subtyping that is required by the elaboration approach. The naive use of coercive subtyping has a significant impact on performance. Moreover, the choice of pairs in the elaboration means that, merge lookup, the most common operation on merges, takes linear time in the worst case.

We show how CP code can be compiled to languages supporting *extensible records* or similar mechanisms. We choose such targets because many existing languages support extensible records or closely related mechanisms like hash maps, which can be dynamically extended with new fields. These data structures are usually highly optimized to enable efficient implementations. This is useful for obtaining fast lookup for merges in CP, which are used to encode dynamic inheritance, as well as to model multi-field records. The lookup is type-based, and we employ a compilation scheme that maps any CP type into a record label, leading to an efficient way to perform lookup on merges. The concrete implementation of the CP compiler targets JavaScript, where records are modeled as JavaScript objects, and record extension is modeled by JavaScript's support for object extension. We also present a number of optimizations and conduct an empirical evaluation to evaluate our implementation of the CP compiler.

In summary, the contributions of this paper are:

- **A compilation scheme for dynamic inheritance and family polymorphism.** We model family-polymorphic dynamic multiple inheritance as nested trait composition via merging in CP. We propose an efficient compilation scheme that translates merges into extensible records, where types are used as record labels to perform lookup on merges. We also identify a class of equivalent types to reduce the number of coercions that are required by subtyping.
- **Mechanized type-safety proofs for the compilation scheme.** We formalize the compilation scheme as an elaboration from the λ_i^+ calculus [Bi et al. 2018; Huang et al. 2021] to a calculus with extensible records called λ_r. We prove that this elaboration is type-safe. Both the elaboration and its type-safety proofs are mechanized using the Coq proof assistant.
- **A compiler for the CP language targeting JavaScript.** We implement a compiler for CP that targets JavaScript, following the ideas of the elaboration into extensible records. In addition, the compiler also implements several other features of CP, which are not formalized, including the support for parametric polymorphism and separate compilation.
- **Several optimizations and an empirical evaluation.** We discuss several optimizations that we employ in the CP compiler and conduct an empirical evaluation to measure their impact. Besides, we benchmark the JavaScript code generated by our compiler together with handwritten JavaScript code.

The Coq proofs, the implementation of the CP compiler, and the benchmark suite are all included in the supplementary materials, which are available at:

https://github.com/yzyzsun/CP-next/tree/toplas

2 Dynamic Inheritance, Overriding, and Type Safety

There are only a few statically typed languages that support first-class classes and dynamic inheritance, among which are `gbeta` [Ernst 2000], TypeScript [Microsoft 2012], Typed Racket [Takikawa et al. 2012], and Wyvern [Lee et al. 2015]. Here we take the most popular one, TypeScript, as the main example to illustrate the challenges of type-safe dynamic inheritance and reveal significant limitations of TypeScript's type system. We will also briefly mention JavaScript and Java to further illustrate concepts related to first-class classes and dynamic inheritance. Discussions about `gbeta`, Typed Racket, and Wyvern can be found in Section 8.

2.1 Class Inheritance and Structural Typing

Classes are the reusable building blocks in most OOP languages. They are reused by inheritance, a mechanism to create a new class (called a *subclass*) based on an existing class (called a *superclass*). Inheritance enables the reuse of implementations of methods or properties that are already provided in the superclass. Furthermore, it is possible to override methods of the superclass with new implementations that are more suitable for the subclass.

To make sure that instances of the subclass can be used in any context where its superclass is expected, there is usually a requirement that the subclass has a *subtype* with respect to its superclass. While inheritance is related to implementations, subtyping is a relation between types. In many programming languages, class definitions `class B extends A {...}` introduce both relations between `A` and `B`: class `B` inherits the implementation from class `A`, and it also introduces a subtyping relation between the types of the two classes. For example, type `B` is required to be a subtype of type `A` in TypeScript:

```
class A {}
```

```
class B extends A {
  m(): number { return 0; }
}
```

Owing to the same reason, when a method in the superclass is overridden, the new method in the subclass must have a subtype. For example, the method `f` in `C` is overridden by the one in `D` below:

```
class C {
  f(x: B): number { return x.m(); }
}
```

```
class D extends C {
  f(x: A): number { return 48; }
}
```

The overriding is type-safe because the latter method has a subtype of the former's. According to the standard subtyping rule for functions, the parameter type is *contravariant*, and the return type is *covariant*. Since `A` is a supertype of `B`, the function type `(x: A) ⇒ number` is a subtype of `(x: B) ⇒ number`.

Bivariant subtyping in TypeScript. Perhaps surprisingly, the following code also type-checks:

```
class E {
  f(x: A): number { return 48; }
}
```

```
class F extends E {
  f(x: B): number { return x.m(); }
}
```

The parameter type of method `f` becomes a subtype of that in the superclass, but it still passes the subtyping check. In other words, TypeScript does *not* follow the standard type-theoretic treatment of function subtyping. Instead, TypeScript allows *bivariant* subtyping for method parameters, where the type of method parameters being overridden can either be a subtype or a supertype of the corresponding type in the superclass method. Bivariant subtyping is a well-known source of type unsoundness. It would lead to a runtime error that could have been prevented statically:

```
const o: E = new F;
o.f(new A)  // Runtime Error!
```

TypeScript developers are aware of this, but they justify the use of bivariant subtyping by large numbers of use cases in the libraries that require this functionality.[3] In essence, TypeScript trades type soundness for flexibility and thus supports a more flexible model of inheritance in some cases.

A type-safe alternative model for structural typing. TypeScript's class model adopts the approach that subclasses always generate subtypes of the superclass. Thus, it retains the familiar model that is common in mainstream nominally typed languages like Java, C#, or Scala, which can be seen as an advantage for attracting programmers from those languages.

However, unlike these mainstream programming languages, TypeScript is *structurally* typed. With structural typing, there is a well-known alternative that would enable the overriding in class `F` to be type-safe. As observed by Cook et al. [1990], inheritance is not subtyping. In the context of a language of classes, this means that sometimes subclasses may not be subtypes of the superclass. In particular, the parameter of a *binary method* [Bruce et al. 1995] is supposed to be an object of the class being defined. In this case, the subclass will covariantly refine the type of the method parameters, and thus detaching inheritance from subtyping can be helpful. Since there is no subtyping relation between subclasses and superclasses in an inheritance-is-not-subtyping approach, the standard contravariant subtyping rule, instead of bivariant subtyping, can be used for function parameters, thus preventing type-safety issues that arise from bivariant subtyping.

[3] https://www.typescriptlang.org/tsconfig#strictFunctionTypes

If TypeScript adopted an inheritance-is-not-subtyping approach instead, then the code for `F` could still type-check, but the subclass `F` would not be a subtype of its superclass `E`. Therefore, the runtime error would be prevented because the line would be rejected with a type error:

```
const o: E = new F;  // Invalid upcast in an inheritance-is-not-subtyping approach!
```

While type-safe, the inheritance-is-not-subtyping approach departs from the conventional model adopted by mainstream languages. So, it could be harder for programmers (especially those used to other mainstream OOP languages) to understand that sometimes subclasses cannot be subtypes. This is perhaps a reason (among others) for TypeScript not adopting this approach. Nevertheless, we adopt a model based on inheritance-is-not-subtyping because it allows a more *flexible* but still *type-safe* form of inheritance.

2.2 Unsafe Overriding with Dynamic Inheritance

TypeScript differs from other mainstream OOP languages in that it also supports dynamic inheritance. Dynamic inheritance brings new type-safety considerations with respect to overriding. These issues are not due to the use of bivariant subtyping and appear to be unknown or undocumented by the TypeScript implementers. Nevertheless, in order to obtain a type-safe design, we must be able to address the type-safety issues that may arise from dynamic inheritance. Thus, the purpose of this subsection is to identify such a problem in TypeScript. We call this problem the **inexact superclass problem**, because it arises from a mismatch between the statically expected type of the superclass and the actual (exact) type of the superclass. In Section 3.3, we will show how this problem can be addressed in a type-safe manner.

Dynamic inheritance in TypeScript. While JavaScript accepts the unsafe overriding in Fig. 1, TypeScript detects the type mismatch between the two methods and rejects the code. For top-level classes and static inheritance, TypeScript's type system is quite standard and rejects many unsafe examples. However, the checks that TypeScript does are insufficient for *dynamic inheritance*, which is recommended by the TypeScript documentation to implement mixins.[4] We illustrate the issue in the program in Fig. 2.

Our example follows the guidelines in the TypeScript documentation to type mixins and first-class classes. First of all, a type `Constructor` is declared to represent a class. Since its return type is an empty object type, the type of every class is a subtype of `Constructor`. In other words, every class can be used as `Base`. The function `Mixin` takes a base class of type `TBase` and returns a new class that extends (or overrides) the base class with the method `m`. Then we obtain class `B` by applying `Mixin` to class `A`. Note that `A` already has a method `m` with a different type, and the other method `n` relies on `m` returning a *string*. However, the subclass returned by `Mixin` overrides `m` with a method that returns a *number* instead. Finally, we instantiate the class `B` and call the method `n`. A runtime error occurs because the method `m` is unexpectedly overridden. In essence, we cannot predict the exact type of the superclass at compile time, so we cannot prevent the unsafe overriding as statically typed languages do for second-class classes and static inheritance.

Constrained mixins. The TypeScript documentation also mentions *constrained mixins*, which provide finer control on the base class. In a constrained mixin, the base class is known to have some methods, which is useful for the subclass to safely rely on those methods being present in the superclass. Constrained mixins are modeled with a generic version of `Constructor`:

```
type GConstructor<T = {}> = new (...args: any[]) ⇒ T;
```

[4] https://www.typescriptlang.org/docs/handbook/mixins.html

```
type Constructor = new (...args: any[]) ⇒ {};

function Mixin<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    m(): number { return 48; }  // If m() exists in Base, that one will be overridden.
  };
}

class A {
  m(): string { return "foobar"; }
  n(): string { return this.m().toUpperCase(); }
}

const B = Mixin(A);     // We use A as Base, which contains m() with a different type.
(new B).n()             // Runtime Error!
```

Fig. 2. **Inexact Superclass Problem:** Dynamic inheritance is type-unsafe in TypeScript.

The generic parameter `T` represents the interface of the base class and defaults to an empty object type. For example, we can define another mixin that relies on a method called `pow`:

```
type Exponentiatable = GConstructor<{ pow: (x: number, y: number) ⇒ number }>;

function AnotherMixin<TBase extends Exponentiatable>(Base: TBase) {
  return class extends Base {
    cube(x: number) { return this.pow(x, 3); }
  };
}
```

In `AnotherMixin`, the method `cube` relies on `this.pow`, which is declared to be present by the interface `Exponentiatable`. Similarly, in the definition of `Mixin` in Fig. 2, we could declare that `TBase` extends some type like `GConstructor<AInterface>`. Although the base class is constrained by the interface now, it still does not help with the issue of unsafe overriding. The problem here is that the base class may contain more methods than the expected interface. For instance, the base class could contain another method called `cube` that would return a *string*, and would be called in the base class by some other method `rubik`. Then we could still run into the same problem, if `rubik` is called from an object that combines both classes. There is no way in TypeScript to express the constraint that `cube` is *absent* in the base class. Such absence constraints are key to preventing unsafe overriding in dynamic inheritance while retaining flexibility.

From static to dynamic inheritance. The crucial point in our examples is that dynamic inheritance has the flexibility to pass a class with a *subtype* of the expected type for the base class in `Mixin`. Languages with static inheritance and second-class classes, like Java or C#, do not have this flexibility. Subclassing is usually modeled with a construct like `class B extends A {...}`. In languages with first-class classes, `A` can be an arbitrary expression; but in languages with static inheritance, it can only be a concrete class name. In Java, for instance, a class `A` is associated both to a type `A`, which is the *exact* interface (or type) of the class, and a corresponding implementation of type `A`. In other words, a class declaration has two roles in these languages: declaring an interface and providing an implementation with exactly that interface. Thus, we can never inherit from an implementation

```
class ANSIString {
  constructor(str) {
    this.length = str.length;
    this.chars = str.split('');
  }
  Iterator() {
    const outer = this;
    return class {
      index = 0;
      hasNext() { return this.index < outer.length; }
      next() { return outer.chars[this.index++]; }
    };
  }
  print() {
    const it = new (this.Iterator());  // Iterator is dynamically bound.
    while (it.hasNext()) alert(it.next());
  }
}
```

Fig. 3. A string iterator in JavaScript using nested classes.

that has a subtype of the superclass type. This avoids the inexact superclass problem that we have to face with dynamic inheritance in our TypeScript example, at the cost of flexibility.

2.3 Nested Classes via First-Class Classes

Both JavaScript and TypeScript support first-class classes: a class can be defined in various places including within another class, or even a method. Thus, *nested classes* come (almost) for free once a language supports first-class classes. In contrast, some other OOP languages, such as Java, do not support first-class classes, but they still add support for nested classes as a separate feature.

Nested classes are useful for encapsulation, and usually, they can make use of the definitions from the outer class. For example, Fig. 3 shows how to model a string-specific iterator as a nested class in JavaScript. The constructor for the `Iterator` class is modeled by a *factory* method. The method `print` relies on the nested iterator class to iterate over the characters in the string.

Why not a class field? In JavaScript, the use of the factory method is important to provide access to **`this`** of the outer class. If we declare a class field directly with `Iterator = class {...}`, we would not be able to access the properties and methods of `ANSIString` within `Iterator`. JavaScript does not provide a direct way to refer to the outer **`this`** from the nested class. That is why we have to capture the reference to the outer **`this`** in a variable `outer` before using it in the nested class. Then, the properties declared in the outer class, `length` for example, can be accessed via `outer.length`. The second reason for using a factory method is to make access to `super.Iterator` possible in a subclass of `ANSIString`. In JavaScript, a class field defined by the superclass is not accessible in the subclass via **`super`**. Declaring `Iterator` as a factory method bypasses the restriction. Although we do not use `super.Iterator` in the current example, Section 2.4 will show some use cases.

Overriding nested classes. In JavaScript, the inheritance behavior for nested classes is consistent with that for methods: they both employ an overriding semantics. This is partly because nested classes must always be accessed via a property or a method, and then we just use the default overriding semantics for them. The ability to override nested classes allows some useful forms of family polymorphism, as we shall discuss in Section 2.4. However, it is also a problem for type safety since we can override a class with another class that has an entirely (or partially) different set of methods. For example, the following code is allowed in JavaScript:

```
class UTF8String extends ANSIString {
  Iterator() { return class {
    forEach(callback) { /* ... */ }
  }; }
}
(new UTF8String("Hi")).print();  // Runtime Error!
```

The class `Iterator` nested in `ANSIString` contains two methods `hasNext` and `next`, while the one nested in `UTF8String` only contains a different method `forEach`. After overriding, `print` triggers a runtime error since it depends on the aforementioned two methods. Therefore, code relying on nested classes having a certain interface can be completely broken by an override that replaces the class with some other incompatible class.

Nested classes in TypeScript. TypeScript also attempts to prevent type-unsafe overriding for nested classes. Similar code will be rejected by TypeScript because of the type incompatibility between the two nested classes. However, with dynamic inheritance, the type system still suffers from similar issues to those shown in Fig. 2:

```
function Mixin<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    Iterator() { return class {
      forEach(callback: (_: string) => void) { /* ... */ }
    }; }
  };
}
const UTF8String = Mixin(ANSIString);
(new UTF8String("Hi")).print();  // Runtime Error!
```

Therefore, TypeScript's support for nested classes is also affected by the inexact superclass problem. Thus, nested classes can have type-soundness issues as well.

Nested classes with shadowing in Java. Finally, let us make a small digression to see how nested classes are treated in Java. Fig. 4 illustrates a variant of our example in Java. Similarly to the example in JavaScript, we create a new class `UTF8String` that inherits from `ANSIString` and define a different set of methods in the nested class `Iterator`. The code type-checks in Java and is still type-safe. The key to the type safety is that, unlike methods, nested classes are not implicitly overridden in Java. Instead, the `Iterator` in `UTF8String` *shadows* the one in `ANSIString`. In other words, `new Iterator()` in `print` is statically bound and is always instantiating `ANSIString.Iterator`. Nested classes are not dynamically dispatched in Java, which is inconsistent with the inheritance behavior for methods. The shadowing approach has the advantage of type safety, but this comes at the cost of flexibility, since the ability to override and dynamically bind nested classes is useful, as we shall see next.

2.4 Virtual Classes and Family Polymorphism

The ability to override or refine nested classes provides a considerable amount of flexibility, and is a key idea behind concepts such as *virtual classes* [Clarke et al. 2007; Ernst et al. 2006; Madsen and Møller-Pedersen 1989] and *family polymorphism* [Ernst 2001; Saito et al. 2008; Zhang and Myers

```
class ANSIString {
    int length;
    char[] chars;
    ANSIString(String str) {
        length = str.length();
        chars = str.toCharArray();
    }
    class Iterator {
        int index;
        boolean hasNext() { return index < length; }
        char next() { return chars[index++]; }
    }
    void print() {
        Iterator it = new Iterator();  // Iterator is statically bound.
        while (it.hasNext()) System.out.print(it.next());
    }
}
class UTF8String extends ANSIString {
    UTF8String(String str) { super(str); }  // We trivially call super's constructor.

    class Iterator {  // This class shadows ANSIString.Iterator.
        void forEach(Consumer<? super Character> action) { /* ... */ }
    }
}
```

Fig. 4. Nested classes in Java, with a shadowing semantics.

2017]. Thus, as we shall argue in this subsection, both JavaScript and TypeScript support virtual classes to a large extent, which can be useful for writing highly modular and reusable code.

Virtual classes. As we have seen before, a method in a superclass can be overridden in a subclass to refine its behavior. A call to the method is dynamically dispatched according to the runtime type of the object. Such a late-bound method is called a *virtual method*. In the same way, the power of dynamic dispatching can be extended to nested classes. *Virtual classes* are nested classes that can be overridden (or rather refined) in subclasses, and the reference to the virtual class is determined by the runtime type of the object of the outer class. Virtual classes were originally introduced in the BETA programming language [Madsen et al. 1993], and they are also essentially supported in JavaScript and TypeScript via first-class classes and the overriding semantics.

Family polymorphism. Virtual classes enable *family polymorphism*, which naturally solves the long-standing dilemma of modularity and extensibility – *the expression problem* [Wadler 1998] – in a Scandinavian style [Ernst 2004]. In the expression problem, the challenge is to provide various operations (evaluation, pretty-printing, etc.) over various expressions (numbers, addition, negation, etc.) in a modular fashion. A satisfactory solution should allow modular, type-safe extension to both expressions and operations.

We start the example with numeric literals and addition, as well as the evaluation operation in Fig. 5a. `Lit`, for numeric literals, and `Add`, for addition, form the initial class family `FamilyEval`.

```
type Eval = { eval: () ⇒ number };

class FamilyEval {
  Lit(n: number) {
    return class {
      eval() { return n; }
    };
  }
  Add(l: Eval, r: Eval) {
    return class {
      eval() { return l.eval() +
                      r.eval(); }
    };
  }
}
```

(a) Initial family.

```
type Print = { print: () ⇒ string };

class FamilyPrint extends FamilyEval {
  Lit(n: number) {
    return class extends super.Lit(n) {
      print() { return n.toString(); }
    };
  }
  Add(l: Eval&Print, r: Eval&Print) {
    return class extends super.Add(l, r) {
      print() { return l.print() + " + " +
                       r.print(); }
    };
  }
}
```

(b) Adding a new operation.

```
class FamilyNeg extends FamilyPrint {
  Neg(e: Eval&Print) {
    return class {
      eval()  { return -e.eval(); }
      print() { return "-(" + e.print() + ")"; }
    };
  }
}
```

(c) Adding a new expression.

Fig. 5. Expression Problem in TypeScript.

Since TypeScript is structurally typed, we do not need to declare an abstract class or interface `Exp` together with `Lit` and `Add`. Instead, we can directly use type `Eval` to annotate the parameters of `Add`.

To add a new operation, say pretty-printing, we can create a new class family `FamilyPrint` that inherits from `FamilyEval`. Fig. 5b shows the code for the new family. In the new family, `Lit` and `Add` also inherit from **`super`**`.Lit` and **`super`**`.Add`, and a new method `print` is added to both of them. The new operation is represented by type `Print`, and the parameters of `Add` are refined to have type `Eval&Print`. As mentioned in Section 2.1, TypeScript allows bivariant subtyping for parameters of class members, so the unusual refinement of `Add` type-checks here. Note that the overriding of nested classes is a special case here: `Lit` and `Add` are simply extended with new methods, with no existing methods being overridden. In other words, the nested classes are being *merged*, instead of overriding existing functionality.

Similarly, we create a new family `FamilyNeg` for a new expression, say negation in Fig. 5c. Finally, we instantiate `FamilyNeg` and build an expression using all three constructors (`Lit`, `Add`, and `Neg`). Both operations (`eval` and `print`) are available for the expression:

```
const fam = new FamilyNeg();
const e = new (fam.Add(new (fam.Lit(48)), new (fam.Neg(new (fam.Lit(2))))));
```

```
e.print() + " = " + e.eval()  // "48 + -(2) = 46"
```

In this way, we can solve the expression problem in TypeScript (modulo the type-safety requirement). Although TypeScript does not fully ensure type safety, its support for a rather minimal encoding of virtual classes allows a lot of flexibility and reuse, which can be quite useful in practice.

One final remark is that the solution in TypeScript is still not completely satisfactory because the order of extensions is fixed by the inheritance hierarchy (from `FamilyEval` to `FamilyPrint` to `FamilyNeg`). In other words, the extension of `Neg` is coupled to the extension of `Print`, and we cannot use the extension of `Neg` independently. This issue was not mentioned by Wadler in the original expression problem, but it was later identified by Zenger and Odersky [2005] as *independent extensibility*. In TypeScript, a possibility to address the coupling issue is to adopt the mixin pattern, making class families such as `FamilyPrint` and `FamilyNeg` functions parametrized by the family superclass. For simplicity of presentation, we have just employed static inheritance here. We will also address this issue in CP's solution in Section 3.4, which provides a simple and natural approach to avoid coupling and, additionally, is type-safe.

2.5 Problem Statement and Paper Roadmap

A three-fold challenge: achieving flexibility, efficiency, and type safety. Ideally, programming languages with dynamic inheritance and first-class classes should have three properties:

(1) **Flexibility:** The language should be flexible so that highly dynamic patterns of inheritance are allowed. Thus, it should be possible to support dynamic forms of mixins or traits, as well as nested classes or even virtual classes and family polymorphism.
(2) **Reasonable efficiency and separate compilation:** For practical implementations, it is desirable to have a compilation model that is reasonably efficient and supports good software engineering properties, such as separate compilation.
(3) **Type safety:** The language should be type-safe, so that type errors can be prevented statically.

Both JavaScript and TypeScript support points (1) and (2) well. As we have seen, with first-class classes, we can model dynamic inheritance, mixins, nested classes, and even virtual classes and family polymorphism. Therefore, the inheritance model provided by JavaScript and TypeScript is expressive and flexible. Furthermore, there has been a lot of work on optimizing JavaScript implementations, so JavaScript and TypeScript's inheritance and class model are reasonably efficient.

Unfortunately, for point (3), TypeScript's support for type-checking first-class classes has a few type-soundness holes. Some of these holes, such as the use of bivariant subtyping, are known and documented. First-class classes bring new issues, such as the inexact superclass problem. The inexact superclass problem can be avoided by moving into a model based on static inheritance, which is the option widely adopted by most mainstream languages. However, this trades flexibility for type safety. Ideally, we want to avoid this trade-off. Retaining flexibility and type safety while addressing the inexact superclass problem is non-trivial. In particular, it seems to be hard with the overriding semantics of JavaScript, which simply overrides properties that have the same name. Thus, to achieve the three goals together, a new compilation scheme seems desirable.

Previous work on compositional programming and CP [Zhang et al. 2021] has addressed points (1) and (3). However, that work has not studied practical implementability questions, such as how to have a reasonably efficient compilation model with separate compilation. Although there is an implementation of the CP language, this implementation is based on an interpreter. Moreover, the semantics underlying compositional programming languages rely on *coercive subtyping* [Luo et al. 2013], which raises immediate questions in terms of efficiency, since upcasts lead to computational overhead. A naive implementation that inserts coercions every time upcasting is needed has a prohibitive cost, which can be *orders of magnitude* slower than JavaScript programs.

Solving the three-fold challenge: efficient compilation for CP. The problem that this paper solves is how to compile compositional programming much more efficiently while also supporting separate compilation. Therefore, we obtain property (2), which was missing on previous work on CP. Thus, we can solve the three-fold challenge. We should emphasize that our work lacks various features supported by JavaScript and TypeScript, and the semantics we employ for inheritance has some important differences from JavaScript. Thus, our work does not offer an immediate solution that TypeScript can adopt as a type-safe replacement for their current class model. Nevertheless, our compilation model can be useful for new languages that aim to have highly expressive models of inheritance while ensuring type safety. Moreover, it can inform existing language designers, who may be able to borrow some ideas to improve their language designs.

Paper roadmap. In Section 3, we will give an overview of CP and see how the CP language addresses the type-safety issues of dynamic inheritance while retaining flexibility. Section 4 then describes the key ideas in our new compilation scheme and its implementation in the CP compiler. Section 5 formalizes a simplified version of the compilation scheme along some of the key ideas. Section 6 explains implementation details, including the JavaScript code that is generated and some core optimizations in the CP compiler. Section 7 provides an empirical evaluation, and Section 8 discusses related work. Finally, Section 9 concludes the paper and outlines future work.

3 Dynamic Inheritance in CP

CP [Zhang et al. 2021] is a statically typed language that supports dynamic inheritance via merging and still guarantees type safety. In this section, we first give an overview of the key features of CP: merges and disjointness. We then show how potential conflicts in dynamic inheritance are resolved in CP, and how CP solves the inexact superclass problem. Finally, we demonstrate a form of *dynamic* family polymorphism in CP.

3.1 Merges, Disjointness, and the Treatment of Conflicts

The *merge operator* is used to construct a term that has an intersection type. The idea originates from the Forsythe programming language by Reynolds [1997], but the general merge operator that we employ was first introduced by Dunfield [2014]. If e_1 has type A and e_2 has type B, then the merged term $(e_1 \,,\, e_2)$ has the intersection type $(A \,\&\, B)$. When we specialize A and B to be record types, $e_1 \,,\, e_2$ is basically concatenating two records. Therefore, the merge operator can be regarded as a generalized form of record concatenation. Since objects are commonly modeled as records in the literature, record concatenation, or more generally, the merge operator is closely related to inheritance [Cook and Palsberg 1989; Wand 1991].

However, adding an unrestricted merge operator to a language would lead to semantic ambiguity. In other words, the semantics of the language would become non-deterministic. For example, $(1 \,,\, 2) + 3$ could evaluate to either 4 or 5. That is why Oliveira et al. [2016] introduced the notion of *disjointness* to avoid ambiguity. If specialized to record types again, disjointness is similar to constraints used in *row polymorphism* [Harper and Pierce 1991]. In the presence of disjointness, the two terms to be merged are restricted to have disjoint types so that the information they convey does not overlap. By this means, $1 \,,\, 2$ is rejected because it is not well-typed, as `Int` and `Int` itself are not disjoint.

Interaction between merging and subtyping. According to the notion of disjointness, `{ x: Int }` and `{ x: Int }` itself are not disjoint either, so the merge `r,s` is rejected in the following code:

```
let merge (r: { x: Int }) (s: { x: Int }) = r,s in  -- Type Error!
merge { x = 1 } { x = 3 }  --> { x = ? }
```

If we further consider subtyping, the merge operator is still problematic, and disjointness alone is not sufficient to prevent ambiguity. For example, consider the following code:

```
let merge (r: { x: Int }) (s: { y: Int }) = r,s in
merge { x = 1; y = 2 } { x = 3; y = 4 }  --> { x = ?; y = ? }
```

Note that we change the type of `s` from `{ x: Int }` to `{ y: Int }`. Although the type of `s` is now disjoint with that of `r`, we can pass terms of their subtypes to `merge`. In this case, `r` has an extra field `y` and `s` has an extra `x`. Now the issue of ambiguity occurs again.

If we look at the function `merge` statically, we would expect that the field `x` is from `r` and `y` from `s`. Therefore, the most reasonable result for the code above is `{ x = 1; y = 4 }`. However, there is no naive way to implement the merge operator to achieve this result. *Neither* left-biased *nor* right-biased overriding is able to handle this case. Furthermore, selecting other fields at run time can lead to type unsoundness. For example, consider a variant of the previous merge:

```
merge { x = 1; y = "Hi" } { x = "Bye"; y = 4 }  --> { x = ?; y = ? }
```

Statically, the function is expected to compute a value of type `{ x: Int; y: Int }`, but fields of type `String` could be selected. The interaction between record concatenation and subtyping is inherently difficult and was the reason preventing Cardelli and Mitchell [1991] from choosing concatenation as the primitive operator in their calculus. This problem is closely related to the inexact superclass problem discussed in Section 2.2, which can be seen as a manifestation of the more general problem identified by Cardelli and Mitchell.

The solution found in the line of work by Oliveira et al. [2016] is to employ a *coercive* semantics of subtyping, where a subtyping relationship $A <: B$ implies a coercion function of type $A \rightarrow B$. This solution picks the field `x` from `s` and `y` from `r`, by being aware of the static types when selecting components. In the previous example, during the function application, `r` is coerced to a single-field record `{ x = 1 }`, corresponding to the parameter type `{ x: Int }`. A similar coercion is inserted for `s` as well, coercing it to `{ y = 4 }`. Then the merge operator simply concatenates `{ x = 1 }` and `{ y = 4 }`, which has no ambiguity. Thus, a combination of disjointness and a coercive approach to subtyping is able to eliminate the ambiguity introduced by an unrestricted merge operator.

Disjoint polymorphism and disjointness constraints. In the previous example, some type information about the records being merged is lost. But we may wish to preserve other fields in the records that do not create ambiguity. This can be achieved by merging polymorphic terms, whose static types are not fully known. For example, consider a variant of the previous example:

```
let mergeSub (A <: { x: Int }) (B <: { y: Int }) (r: A) (s: B) = r,s in
mergeSub @{ x: Int; y: Int } @{ x: Int; y: Int } { x = 1; y = 2 } { x = 3; y = 4 }
```

The code is written in pseudo-CP, where `<:` denotes the upper bound of a type parameter. In this example, `A` and `B` are declared to be subtypes of `{ x: Int }` and `{ y: Int }` respectively. Since CP does not yet support implicit polymorphism, both type parameters are instantiated explicitly on the second line. Like in Haskell, `@` is the prefix of type arguments in CP. With bounded quantification [Cardelli and Wegner 1985], we cannot guarantee the disjointness of `A` and `B`, so the issue of ambiguity comes back again. This issue can be solved by *disjoint quantification* [Alpuim et al. 2017] (disjointness is denoted by $*$):

```
let mergeDis (A * { y: Int })    (B * A & { x: Int })  -- B * A and B * { x: Int }
             (r: A & { x: Int }) (s: B & { y: Int }) = r,s in
mergeDis @{ y: Int } @{ x: Int } { x = 1; y = 2 } { x = 3; y = 4 }  -- Type Error!
mergeDis @Top @Top { x = 1 } { y = 4 }  --> { x = 1; y = 4 }
```

Note that the type of `r` is now `A & { x: Int }` instead of `A`. This is how we usually translate subtype-bounded quantification to disjoint quantification [Xie et al. 2020]. The type parameter `A` is declared to be disjoint with `{ y: Int }` to avoid the overlap, and `B` is disjoint with `{ x: Int }` similarly. Another important constraint here is the disjointness of `A` and `B`, ensuring that other fields will never conflict as well. For example, consider a third field of type `{ z: Int }`:

```
mergeDis @{ z: Int } @{ z: Int } { x = 1; z = 5 } { y = 4; z = 6 }  -- Type Error!
mergeDis @Top @{ z: Int } { x = 1 } { y = 4; z = 6 }  --> { x = 1; y = 4; z = 6 }
```

The first line of code fails to type-check because `A` and `B` are not disjoint and both contain a field of type `{ z: Int }`. The second line resolves the conflict, and we can access all three fields after merging. The absence of certain fields is not expressible in TypeScript. As we shall see in Section 3.3, this is important for CP to safely handle dynamic inheritance.

3.2 From Merging to Inheritance

Let us now turn to the topic of how we model inheritance as merging. According to the denotational semantics of inheritance [Cook and Palsberg 1989], an object is essentially a record, and a class (or a trait in CP) is essentially a function over records. Also note that, since CP is a purely functional language, there is no distinction between object fields and methods – a method is just a field that may have a function type. Class `A` in Fig. 2 can be encoded as:

```
type Rcd = { m: String; n: String };

-- class A
mkA = \(this: Rcd) → { m = "foobar"; n = toUpperCase this.m };
```

The function parameter `this` is a self-reference. With the self-reference, we can refer to other fields like `this.m` in the `n` field. In this model, the instantiation of a class is obtained by taking a fixpoint of the function. Furthermore, class inheritance can be encoded as record concatenation:

```
-- class B extends A
mkB = \(this: Rcd) → let super = mkA this in super , { m = 48 };
```

We first provide the new self-reference to `mkA` to obtain `super`. Then we merge `super` with the body of class `B` to obtain the final object. After instantiating class `B` with a fixpoint, we can access the `n` field:

```
o = fix this: Rcd. mkB this;  --> { m = "foobar"; n = "FOOBAR"; m = 48 }
o.n  --> "FOOBAR"
```

Here we get the expected result instead of a runtime error. The key point is that we allow duplicate labels as long as the fields have disjoint types. Because of the merging semantics of CP, `o` will have two `m` fields: one of type **`Int`** and the other of type **`String`**. Thus, unlike TypeScript, no implicit (and type-unsafe) overriding happens in this case. Instead, both `{ m = "foobar" }` and `{ m = 48 }` are kept in the record `o`, and `toUpperCase this.m` will automatically pick the former one. Internally, `o.m` has the intersection type **`String&Int`**, which means it contains a merge of a string and an integer. Such behavior is a kind of *overloading by return type*, which is supported in some languages such as Swift and Haskell (via type classes) [Marntirosian et al. 2020].

Traits in CP follow the aforementioned model of inheritance. Therefore, the example above can be rewritten in the form of traits:

```
mkA = trait [this: Rcd] ⇒ { m = "foobar"; n = toUpperCase this.m };
mkB = trait [this: Rcd] inherits mkA ⇒ { m = 48 };
```

The self-type annotation `[this: Rcd]` corresponds to the function parameter `this` in the previous code. If there is no use of `this` in any field, the self-type annotation can be omitted. The instantiation of a trait is more conveniently done by the **new** keyword:

```
o = new mkB;  o.n  --> "FOOBAR"
```

Merging versus overriding. So far we have discussed how disjointness prevents ambiguity in merging. Basically we avoid any overlap between the two terms to be merged. According to the model of inheritance that we use, this constraint automatically applies to inheritance as well. One may ask whether this means that overriding is forbidden in CP. This is not true: programmers can explicitly declare overriding using the **override** keyword. For example, we can have:

```
base = trait ⇒ { m = 48 };
derived = trait inherits base ⇒ { override m = super.m - 2 };
```

This forces programmers to think about the potential conflict and make a decision. Scala and other programming languages also require programmers to write **override** explicitly. Accidentally overriding a field or a method in the base class can lead to unexpected behavior, which is a common source of bugs in OOP languages. For example, `base` may have other fields that assume `m` is exactly 48 and will not work properly if `m` is overridden. This issue is also known as the *fragile base class problem* [Mikhajlov and Sekerinski 1998]. At run time, CP will exclude the overridden field from the base trait before merging.

Multiple inheritance. CP supports a form of multiple trait inheritance, which makes the treatment of conflicts more complicated. For example, consider the following code:

```
base1 = trait ⇒ { m = 48; n = "Hi" };
base2 = trait ⇒ { m = 46; n = "Bye" };
derived = trait inherits base1 , base2 ⇒ { ... };  -- Type Error!
```

In some OOP languages that support multiple inheritance, such as Scala and Python, the order of inheritance determines which field is chosen if fields in different base classes have the same name. However, the default resolution order may not be what programmers desire. It easily causes bugs if programmers are not aware of the implicit overriding. What is worse, there is no way to pick `n` from `base1` and `m` from `base2` at the same time. In CP, programmers are again required to explicitly resolve the conflicts, while having more flexibility to choose the desired fields:

```
derived = trait inherits base1\m , base2\n ⇒ { ... };  -- OK!
      --> trait ⇒  { n = "Hi" } , { m = 46 } , { ... }
```

With the record restriction operator (\) powered by *type difference* [Xu et al. 2023], we can easily remove `m` from `base1` and `n` from `base2`. In traditional OOP languages, inheritance involves two things: inheriting all fields from the base classes, and overriding some of them. In contrast, symmetric merging in CP does not imply any overriding. Nevertheless, for the sake of convenience, CP also provides biased versions of merging (e.g. `base1 ,+ base2` or `base1 +, base2`) if left-to-right or right-to-left overriding is desired. They are also powered by type difference under the hood.

3.3 Dynamic Inheritance in CP

Now let us go back to the safety issue demonstrated in Fig. 2 and see how it can be solved in CP. The code for a CP solution is shown in Fig. 6. Here the function `mixin` has two parameters: `TBase` is a type parameter, which is disjoint with `{ m: Int }`; and `base` is a term parameter, which is a trait that implements `TBase`. Like first-class classes in TypeScript, we can dynamically create a trait that inherits from `base` in CP. The difference here is that we can declare the absence of

```
mixin (TBase * { m: Int }) (base: Trait<TBase>) =
  trait [this: TBase] inherits base ⇒ { m = 48 };

mkA = trait [this: { m: String; n: String }] ⇒ {
  m = "foobar";
  n = toUpperCase this.m;
};

o = new mixin @{ m: String; n: String } mkA;
o.n  --> "FOOBAR"
```

Fig. 6. Solving the inexact superclass problem in CP.

`{ m: Int }` in the trait `base` to make sure that there is no conflict. As mentioned in Section 3.2, CP does a fine-grained disjointness check that considers, not only the label name, but also the field type. Therefore, `{ m: String }` is disjoint with `{ m: Int }`, and there is no conflict in the dynamic inheritance. Since both versions of `m` fields are available in `o`, the `n` field can still rely on the original `m` field that contains a string. Together with disjointness constraints, type safety is guaranteed in CP without sacrificing the flexibility of dynamic inheritance.

Finally, if we apply `mixin` to a different trait that contains an `m` field of type **`Int`**:

```
mkA' = trait ⇒ { m = 0; n = 0 };
o = new mixin @{ m: Int; n: Int } mkA';  -- Type Error!
```

We will get a type error because `{ m: Int; n: Int }` is *not* disjoint with `{ m: Int }`. In other words, the field `m` in `mkA'` conflicts with `m` in `mixin`.

3.4 Family Polymorphism in CP

Here we revisit the example of family polymorphism in Section 2.4 and show how it can be implemented in CP. As before, we start with the evaluation of numeric literals and addition. The CP code is shown in Fig. 7a. The compositional interface `AddSig` serves as the specification of expressions, while type `Eval` represents the evaluation operation. Note that `<Exp>` is a special type parameter called a *sort* in CP. A sort is kept abstract until it is instantiated with a concrete type like in `AddSig<Eval>`. The interface `AddSig<Eval>` is implemented by trait `familyEval`, where syntactic sugar called *method patterns* is used to keep code compact. The desugared code is:

```
familyEval = trait implements AddSig<Eval> ⇒ {
  Lit = \n   → trait ⇒ { eval = n };
  Add = \l r → trait ⇒ { eval = l.eval + r.eval };
};
```

Although the syntactic sugar makes it seem that `eval` is defined by pattern matching of constructors, `(Lit n)` and `(Add l r)` are actually nested traits, which are virtual and can be refined in CP.

The solution to the expression problem in CP is quite straightforward. To extend operations, we instantiate the sort with another type and implement it with another trait. For example, Fig. 7b shows how to add support for pretty-printing. In the other dimension, we add negation to numeric literals and addition. We define a new compositional interface and implement both operations with a trait in Fig. 7c. This time we instantiate the sort of `NegSig` with the intersection type `Eval&Print`.

Finally, we can compose the two-dimensional extensions together by the merge operator easily:

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

type Eval = { eval: Int };

familyEval =
  trait implements AddSig<Eval> ⇒ {
    (Lit   n).eval = n;
    (Add l r).eval = l.eval + r.eval;
  };
```

(a) Initial family.

```
type Print = { print: String };

familyPrint =
  trait implements AddSig<Print> ⇒ {
    (Lit   n).print = toString n;
    (Add l r).print = l.print ++ " + "
                   ++ r.print;
  };
```

(b) Adding a new operation.

```
type NegSig<Exp> = { Neg: Exp → Exp };

familyNeg =
  trait implements NegSig<Eval&Print> ⇒ {
    (Neg e).eval  = -e.eval;
    (Neg e).print = "-(" ++ e.print ++ ")";
  };
```

(c) Adding a new expression.

Fig. 7. Expression Problem in CP.

```
fam = new familyEval,familyPrint,familyNeg : AddSig<Eval&Print> & NegSig<Eval&Print>;
```

Nested composition and distributive subtyping. The merge of the three traits seems simple from a syntactic perspective. However, it requires a more sophisticated mechanism under the hood. Let us look at the desugared code for the merge between `familyEval` and `familyPrint`:

```
trait implements AddSig<Eval> ⇒ {                  -- familyEval
  Lit = \n   → trait ⇒ { eval = n };
  Add = \l r → trait ⇒ { eval = l.eval + r.eval };
} ,
trait implements AddSig<Print> ⇒ {                 -- familyPrint
  Lit = \n   → trait ⇒ { print = toString n };
  Add = \l r → trait ⇒ { print = l.print ++ " + " ++ r.print };
}
```

Our expectation is that the result of merging should contain, for example, a single constructor `Lit` that supports both the `eval` and `print` operations. Therefore, the result should be equivalent to:

```
trait implements AddSig<Eval&Print> ⇒ {
  Lit = \n   → trait ⇒ { eval = n;
                         print = toString n };
  Add = \l r → trait ⇒ { eval = l.eval + r.eval;
                         print = l.print ++ " + " ++ r.print };
}
```

To achieve this, CP employs *nested composition* [Bi et al. 2018] and *distributive subtyping* [Barendregt et al. 1983], where traits, records, and functions distribute over intersections. In other words, merging applies to the whole trait hierarchy, including nested traits. This example showcases family polymorphism by the refinement of nested traits (i.e. CP's version of virtual classes).

With these features available in CP, we can access the three constructors (`Lit`, `Add`, and `Neg`) as well as the two operations (`eval` and `print`), similarly to the previous TypeScript code:

```
e = new fam.Add (new fam.Lit 48) (new fam.Neg (new fam.Lit 2));
e.print ++ " = " ++ toString e.eval  --> "48 + -(2) = 46"
```

Dynamic family polymorphism. Since merging generalizes dynamic inheritance, we can rewrite `familyNeg`, for instance, using a mixin style:

```
familyNeg (TBase * NegSig<Eval&Print>) (base: Trait<TBase>) =
  trait [this: TBase] implements NegSig<Eval&Print> inherits base ⇒ {
    (Neg e).eval  = -e.eval;
    (Neg e).print = "-(" ++ e.print ++ ")";
  };
fam = new familyNeg @AddSig<Eval&Print> (familyEval,familyPrint)
    : AddSig<Eval&Print> & NegSig<Eval&Print>;
```

By applying `familyNeg` to `(familyEval,familyPrint)`, we dynamically create a trait that inherits from the latter. Of course, we can choose other traits as a base trait at run time, which is supported by dynamic inheritance in CP.

Note that in Section 2.4, `FamilyEval`, `FamilyPrint`, and `FamilyNeg` have a statically fixed inheritance hierarchy. As a result, the negation expression cannot be separated from the other two expressions because `FamilyNeg` is a subclass of `FamilyPrint`. In contrast, the inheritance hierarchy can be dynamically determined in CP, so `familyEval`, `familyPrint`, and `familyNeg` can all be individually used or composed with any other traits. In fact, CP's solution solves a dynamic variant of the expression problem, which can be seen as the combination of the expression product line [Lopez-Herrejon et al. 2005] and dynamic software product lines [Hallsteinsen et al. 2008].

3.5 Discussion

In this and the previous section, we have seen that both CP and JavaScript/TypeScript support a powerful and expressive form of dynamic inheritance. However, there are some important differences worth noting:

- **CP is type-safe.** While the three languages provide a high degree of flexibility, CP is the only language which combines flexibility and type safety.
- **No implicit overriding in CP.** Unlike JavaScript/TypeScript, where implicit overriding is common, CP adopts a trait model, so implicit overriding can *never* happen.
- **Dealing with conflicts using disjoint types.** In JavaScript/TypeScript, method overriding is based on *names*. So even when the method or field in the superclass has a different (or disjoint) type, overriding happens when the subclass has a method with the same name. As we have seen, this is the source of type unsoundness in the inexact superclass problem. In CP, methods with disjoint types can coexist in the same object. Thus, for the same situation, CP will not override but inherit the method from the superclass.

These differences are important to obtain flexibility while preserving type safety. However, these differences also mean that the dynamic semantics of CP needs to be different from that of JavaScript/TypeScript. In particular, the dynamic semantics of CP has to be aware of types, since types play a role in determining whether conflicts exist or not, and in unambiguously performing method lookup. This creates important challenges in obtaining an efficient implementation, which have not been addressed in previous work.

4 Key Ideas of the CP Compiler

We now introduce the key ideas under the hood of the CP compiler and describe why and how to compile CP to extensible records in general. We also discuss the major challenges that we had to overcome. Although our implementation targets JavaScript, the design can be adapted to any other language that supports some kind of extensible records. We refer the reader to Section 5 for a formal description of our compilation scheme and Section 6 for the details of our implementation targeting JavaScript.

4.1 Dunfield's Elaboration Semantics

In previous work by Dunfield [2014] and its follow-up work by Oliveira et al. [2016], the semantics of the merge operator is well studied. According to the *non-deterministic* operational semantics given by Dunfield, a merge $48 \,{,}{,}\, \mathbf{true}$ may reduce to 48 or **true**; both are valid reductions. However, such reductions may not preserve types. For instance, in a context like $(48 \,{,}{,}\, \mathbf{true}) - 2$, the merge should reduce to an integer. Alternatively, Dunfield proposes an elaboration semantics into a target calculus with pairs, which is also used by Oliveira et al. Within this framework, an intersection type $A \,\&\, B$ is elaborated into a product type $A \times B$, and a merge $e_1 \,{,}{,}\, e_2$ is elaborated into a pair $\langle e_1, e_2 \rangle$. While an elaboration to pairs offers a simple model for merges, it also imposes significant runtime overhead. We identify three limitations in previous work.

Indirect coercions. Following the elaboration model to pairs, $(48 \,{,}{,}\, \mathbf{true}) - 2$ should be elaborated into $\langle 48, \mathbf{true} \rangle.\mathrm{fst} - 2$. That is, we need to select the first element from the elaborated pair to obtain a well-typed expression. Merges, due to their flexible nature, do not have an explicit elimination form. Then how can we determine where to insert ".fst"? In a type-directed elaboration, we can generate coercion functions according to subtyping judgments in the typing derivation. A rule DTyp-Sub can be found in previous work.

$$\textsc{DTyp-Sub}\quad \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon \qquad A <: B \rightsquigarrow c}{\Gamma \vdash e \Leftarrow B \rightsquigarrow c\,\epsilon} \qquad\qquad \textsc{Ela-Sub}\quad \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \epsilon_1 : A <: B \rightsquigarrow \epsilon_2}{\Gamma \vdash e \Leftarrow B \rightsquigarrow \epsilon_2}$$

The rule DTyp-Sub means that if a source term e is inferred to have type A and elaborated into a target term ϵ, and the subtyping judgment $A <: B$ implies the coercion function c, then e can be checked against type B and elaborated into $c\,\epsilon$. In the aforementioned example, the merge of type $\mathbf{Int} \,\&\, \mathbf{Bool}$ is cast to type $\mathbf{Int}$. Therefore, a coercion function should be implicitly inserted for the subtyping relation $\mathbf{Int} \,\&\, \mathbf{Bool} <: \mathbf{Int}$. Since the latter is the first half of the former, the coercion $\lambda x.\, x.\mathrm{fst}$ is inserted.

A careful reader may notice that rule DTyp-Sub does not produce $\langle 48, \mathbf{true} \rangle.\mathrm{fst} - 2$ as we expect. Instead, it produces $((\lambda x.\, x.\mathrm{fst})\ \langle 48, \mathbf{true} \rangle) - 2$, which is less efficient as it introduces a spurious application. To fill the gap, we propose an alternative rule Ela-Sub in our work and a novel coercive subtyping judgment, which directly coerces ϵ_1 into ϵ_2. In the aforementioned example, the subtyping relation $\mathbf{Int} \,\&\, \mathbf{Bool} <: \mathbf{Int}$ will coerce $\langle 48, \mathbf{true} \rangle$ to produce the more efficient $\langle 48, \mathbf{true} \rangle.\mathrm{fst}$. Although we only avoid one step of beta reduction in this case, a more complicated subtyping judgment will lead to many coercion functions composed together and introduce many spurious applications.

Linear merge lookup. A second important drawback of Dunfield's approach is the representation of merges as nested pairs. The merge operator composes expressions in a binary manner, so extracting one component from nested merges of n components requires $n-1$ projections in the worst case. For example, when adding one more element to the previous merge, $48 \,,\, \textbf{true} \,,\, \texttt{'a'}$ for example, one more projection must be added to the elaborated result as well: $\langle\langle 48, \textbf{true}\rangle, \texttt{'a'}\rangle.\text{fst}.\text{fst} - 2$. Note that we have simplified the coercion application from $((\lambda x.\, x.\text{fst}) \circ (\lambda x.\, x.\text{fst}))\, \langle\langle 48, \textbf{true}\rangle, \texttt{'a'}\rangle$ to $\langle\langle 48, \textbf{true}\rangle, \texttt{'a'}\rangle.\text{fst}.\text{fst}$. Compared with array access or dictionary lookup, such projections are more expensive in terms of both code length and runtime performance.

Pairs are order-sensitive. What is worse, a representation based on pairs has another disadvantage: unnecessary coercions are never optimized. Consider $48 \,,\, \textbf{true}$ and $\textbf{true} \,,\, 48$. These two merges are equivalent in any context. Although they lead to a different order in the elaborated pairs, permutation of components does not matter as long as it is consistent with the projection. For example, $\langle 48, \textbf{true}\rangle.\text{fst} - 2$ is the same as $\langle \textbf{true}, 48\rangle.\text{snd} - 2$. However, permutation can lead to expensive coercions. To cast $48 \,,\, \textbf{true} \,,\, \texttt{'a'}$ to type $\textbf{Char} \,\&\, \textbf{Int} \,\&\, \textbf{Bool}$, every single component needs to be extracted and rearranged:

$$\textbf{let}\ e = \langle\langle 48, \textbf{true}\rangle, \texttt{'a'}\rangle\ \textbf{in}\ \langle\langle e.\text{snd}, e.\text{fst}.\text{fst}\rangle, e.\text{fst}.\text{snd}\rangle$$

Thus, it is desirable to replace nested pairs with other representations that support more efficient merge lookup and avoid conversions between equivalent types.

4.2 Our Representation of Merges

Prologue: compiling overloaded functions. In programming languages that support function overloading, C++ for example, the compiler generates different names for overloaded functions. This process is usually called *name mangling*. If we have a function `f` with two overloaded versions:

```
void f(int x)  { ... }  // f → __Z1fi
void f(bool x) { ... }  // f → __Z1fb
```

Two different names are generated based on the parameter types: the postfix `i` in `__Z1fi` is short for **`int`** and `b` in `__Z1fb` for **`bool`**. After name mangling, the overloaded versions are disambiguated, and the linker can easily associate each call site with a specific version.

Key idea: compiling merges to type-indexed records. When it comes to merging, the situation is similar: a merge contains “overloaded” terms of different types. For example, the merge $48 \,,\, \textbf{true}$ contains both an integer and a boolean value. When compiling the merge, we adopt a similar technique to name mangling. We generate a unique name for every type, which is used to look up the corresponding component. More specifically, a merge is compiled to a record, and the components of the merge become its fields. For example, $48 \,,\, \textbf{true}$ will compile to $\{\texttt{int} \Rrightarrow 48; \texttt{bool} \Rrightarrow \textbf{true}\}$. The labels in the record, which we call *type indices*, are generated from the type of each term. As for nested merges, we also flatten them in one record. Instead of the nested pairs $\langle\langle 48, \textbf{true}\rangle, \texttt{'a'}\rangle$, $48 \,,\, \textbf{true} \,,\, \texttt{'a'}$ is translated into a record of three fields: $\{\texttt{int} \Rrightarrow 48; \texttt{bool} \Rrightarrow \textbf{true}; \texttt{char} \Rrightarrow \texttt{'a'}\}$. The disjointness constraint on merging ensures that the components of a merge have non-overlapping types, hence the fields of the elaborated record are conflict-free (e.g. a merge cannot contain both 48 and 46). The idea of using labels based on types is similar to *type-indexed rows* [Shields and Meijer 2001], though their type system does not involve subtyping at all.

The record design significantly reduces the cost of projections. For $48 \,,\, \textbf{true} \,,\, \texttt{'a'}$, we would not need to project twice to find the exact position when selecting the integer. With a single projection, a component in an n-level merge can be extracted. Besides, record fields are order-irrelevant, which allows us to treat permuted intersection types equivalently. Using our approach, coercing a term from type $\textbf{Int} \,\&\, \textbf{Bool} \,\&\, \textbf{Char}$ to type $\textbf{Char} \,\&\, \textbf{Int} \,\&\, \textbf{Bool}$ has *no cost*, because the elaborated record does not change. In other words, $\{\texttt{int} \Rrightarrow 48; \texttt{bool} \Rrightarrow \textbf{true}; \texttt{char} \Rrightarrow \texttt{'a'}\}$

and $\{\texttt{char} \Rrightarrow \text{‘a’}; \texttt{int} \Rrightarrow 48; \texttt{bool} \Rrightarrow \textbf{true}\}$ are equivalent. In CP, multi-field record types are also represented as intersection types. For example, $\{\ell_1 : \textbf{Int}; \ell_2 : \textbf{Int}\}$ is syntactic sugar for $\{\ell_1 : \textbf{Int}\} \,\&\, \{\ell_2 : \textbf{Int}\}$. Therefore, the order of fields in a record type does not matter either. We will develop a comprehensive theory that accounts for type equivalence and handles all possible cases next.

4.3 Reducing Coercions for Equivalent Types

Coercive subtyping is inevitable in CP, so the performance penalties caused by coercions cannot be neglected. Following the line of discussion above, an important optimization that we identify is to avoid coercions for subtyping between equivalent types, whose impact will be benchmarked in Section 7.1. In our translation scheme, some syntactically different types are translated to the same type index. These types that are treated equivalently after compilation are called *equivalent types* (denoted by $A \fallingdotseq B$). The design of equivalent types is inherently determined by the fact that we represent merges as records. We do not need to distinguish two types after compilation if their terms are compiled to records of exactly the same shape. The most interesting types in our compilation scheme are:

- *Top-like types* [Oliveira et al. 2016], which correspond to empty records because they do not convey any information.
- *Intersection types*, which correspond to multi-field records. Generally speaking, records are order-irrelevant and contain no duplicate labels (or duplicate labels are allowed but fields with the same label have equivalent values).

Considering the characteristics of our record-based representation, we can first derive that all top-like types are equivalent. In addition, two intersection types are considered equivalent if and only if they are formed using any combination of the following three criteria:

- They are permutations of the same set of types, or
- They are equivalent after deduplicating type components, or
- They are equivalent after removing top-like components.

The rules for other types are structural, ensuring that the type equivalence is a congruence.

Although we work hard to reduce the number of coercions, coercions cannot be fully eliminated. Next, we will explain the reason why they are still necessary to CP.

4.4 Necessity of Coercions

In CP, our interpretation of subtyping is *coercive* [Luo et al. 2013], in contrast to the inclusive (also called subsumptive) view of subtyping. That is, a value of a subtype is not a value of a supertype directly, but it contains sufficient information so that it can be converted into a value of a supertype. Such conversions are generated by subtyping derivations and are inserted by the subsumption rule during type checking.

The need for coercive subtyping in CP mainly comes from the unambiguity constraint on merging, for which the redundant information in expressions could be harmful. For example,

$$\textbf{let}\ x = 48 \,,\, \textbf{true}\ \textbf{in}\ \textbf{not}\ (x : \textbf{Int} \,,\, \textbf{false})$$

can evaluate to both **true** and **false** if the boolean component in x is kept. During typing, we use disjointness checks to ensure the static types of the components to be merged (**Int** and **Bool** in this example) do not overlap. But the soundness of such checks is based on the assumption that any expression's dynamic type corresponds to its static type. That is, $x : \textbf{Int}$ should contain nothing other than an integer at run time. So we have to coerce x from $\{\texttt{int} \Rrightarrow 48; \texttt{bool} \Rrightarrow \textbf{true}\}$ to a record that only contains the integer field. With some simplification, the whole expression should compile to:

$$\textbf{let } x = \{\texttt{int} \Mapsto 48; \texttt{bool} \Mapsto \textbf{true}\} \textbf{ in not } (\{\texttt{int} \Mapsto x.\texttt{int}\} +\!\!+ \{\texttt{bool} \Mapsto \textbf{false}\}).\texttt{bool}$$

where $+\!\!+$ denotes runtime record concatenation, which is a key feature of extensible records. In summary, there is a strong correspondence between the value and its static type in CP. So we can directly tell from the declared type how many fields the compiled record has and what the labels are. This design resolves the issue of interaction between merging and subtyping in Section 3.1 and is key to the type safety of dynamic trait inheritance.

Distributive subtyping. Normally, coercions are just removing redundant fields from a compiled record. For example, we coerce x from $\{\texttt{int} \Mapsto 48; \texttt{bool} \Mapsto \textbf{true}\}$ to $\{\texttt{int} \Mapsto 48\}$ in the previous example. This is because a supertype of an intersection type consists of part of the component types, so the compiled record of the supertype contains a subset of the original fields. However, the situation becomes complicated in the presence of *distributive* subtyping. For example, a function of type $(\top \to \textbf{Int}) \,\&\, (\top \to \textbf{Bool})$ can be coerced to type $\top \to \textbf{Int} \,\&\, \textbf{Bool}$ because the former is a subtype of the latter via distributivity. The coercion is not removing fields but merging two functions into a single one.

Let us consider a more practical example based on the expression problem in Section 3.4. Here is a *simplified* version of what happens to the constructors for numeric literals when we compose the evaluation and pretty-printing operations:

```
ep = { Lit = \n → { eval = n } } , { Lit = \n → { print = toString n } };
-- : { Lit: Int → Eval }          & { Lit: Int → Print }
```

As the intersection type indicates, `ep` should compile to a two-field record: one field stores the constructor for `Eval` and the other for `Print`. According to the subtyping relation, via distributivity, it can be used as if it has type `{ Lit: Int → Eval&Print }`:

```
ep.Lit 48  --> { eval = 48; print = "48" }
```

However, such usage expects that the compiled record from `ep` only has one field, whose label corresponds to `{ Lit: Int → Eval&Print }`. Unfortunately, as we showed before, the compiled record actually contains two different labels from the expected one, so the subtyping does not automatically work. That is why we need to insert a coercion here to convert the two-field record to a new one with one single field, which is similar to the previous example of merging two functions.

Our compilation scheme is designed to avoid coercions as much as possible. The aforementioned coercion is not inserted for *direct* usage of record projections or function applications. Instead, the compiled code will select the two functions from the two fields for `ep.Lit` and apply both to 48. The results are then combined into a record so that both `eval` and `print` fields are present.

4.5 Implementation in JavaScript

The extensible records that we have been mentioning are an abstract data type that supports construction, concatenation, and projection. They do not imply any concrete data structure in any particular programming language. They can be implemented as hash tables, binary search trees, or even association lists, and most mainstream languages have built-in and highly optimized support for these data structures. In our implementation, extensible records are implemented as *JavaScript objects*, whose underlying data structure still varies among JavaScript engines. Nevertheless, one thing we are certain of is that accessing properties of an object, which corresponds to record projection in our terminology, is highly optimized in the various engines.

The CP compiler supports modular type checking and separate compilation. In other words, compiling a CP file does not require access to the source code of the libraries that it depends on. What is needed is only the header files of the libraries, which mainly contain type information. Separate compilation largely decreases the rebuilding time since it avoids recompiling its dependencies, and it allows closed-source distribution of libraries. More details about the implementation of separate compilation can be found in Section 6.6.

Type indices. In our implementation, type indices are represented by JavaScript strings (hereinafter, `"string"` is in violet and monospaced). Below is how we represent different types:

- Primitive types are simply represented by their names, e.g. `"int"` for **Int**.
- Function types are represented by their return types, e.g. `"func_int"` for **String** → **Int**.
- Record types are represented by both labels and field types, e.g. `"rcd_l:int"` for $\{\ell : \mathbf{Int}\}$.
- Intersection types are represented by joining the representations of their components after alphabetical sorting, deduplication, and removal of top-like types, e.g. `"(bool&int)"` for $\mathbf{Int} \,\&\, \mathbf{Bool} \,\&\, \top \,\&\, \mathbf{Bool}$. Note that such type indices only occur when intersection types are nested within functions or records. A top-level intersection corresponds to a multi-field record, which has separate type indices for each component.

The representation for function types may be a bit surprising. It originates from the disjointness rule for function types: two function types are disjoint if and only if their return types are disjoint (rule D-ArrowArrow). This rule is derived from the specification of disjointness (Theorem 5.7), which basically means that two disjoint types do not overlap on any meaningful types. For example, `Int → Int` and `Bool → Int` shares a common supertype `Int&Bool → Int`, so these two types are *not* disjoint. If those types are considered to be disjoint, we could have the following application:

```
((\(x: Int) → x + 1),(\(x: Bool) → if x then 1 else 0)) (1,false)
```

Note that both functions can be selected, and we get either 2 or 0 depending on which function we pick. The semantics would be ambiguous in this way. Thus, allowing such merges is unsafe. That is why `Int → Int` and `Bool → Int` are not disjoint, and `"func_int"` cannot occur twice. The disjointness checks in CP rule out the possibility of type index conflicts between two functions in a merge. Our design that includes only return types also avoids very long property names in JavaScript, which may lead to performance issues.

Compiling parametric polymorphism. As we have discussed previously, dynamic inheritance and family polymorphism are already difficult to handle. In those examples, parametric polymorphism also plays an important role, yet we have not mentioned the difficulty of compiling it. The reason why this feature is challenging to compile is a bit more technical: it relates to when to build type indices, namely the labels of the compiled records.

For non-polymorphic types, the labels remain fixed throughout the program execution. However, for polymorphic types, we have to deal with *type instantiation*. For example, we may have a source type `{ f : A → A }`, where the type `A` is a type variable. After the instantiation of `A`, we may have the type `{ f : Int → Int }` or perhaps the type `{ f : Bool → Bool }`. The problem is that different instantiations of polymorphic type variables will produce different labels. So for polymorphic types, the labels cannot be statically computed. To solve this problem, first-class labels [Leijen 2004] are needed so that polymorphic instantiation can build a label at run time and propagate the label that corresponds to the instantiated type. A more detailed explanation with examples can be found in Section 6.2.

Important optimizations. In our implementation, we have applied several optimizations to improve the performance of the generated JavaScript code. Besides the elimination of redundant coercions based on equivalent types in Section 4.3, some important optimizations are:

(1) Reducing intermediate objects using destination-passing style [Shaikhha et al. 2017];

(2) Reducing object copying by detecting whether the compiled term is part of a merge;
(3) Limiting lazy evaluation to certain trait fields to improve performance;
(4) Preventing primitive values from boxing/unboxing;
(5) Avoiding the insertion of coercions for record projections.

These optimizations will be elaborated with examples in Section 6, and their impact on performance will be evaluated in Section 7.1. Among the five optimizations, the last one (5) is formalized.

5 Formalization of the Compilation Scheme

To demonstrate and validate the key ideas of the compilation scheme, this section introduces two calculi for the source and the target languages, respectively, and the elaboration between them.

The *source calculus* is a variant of λ_i^+ [Bi et al. 2018; Huang et al. 2021], which mainly omits parametric polymorphism from F_i^+ [Bi et al. 2019; Fan et al. 2022], the core calculus for CP. Polymorphism is supported in our compiler, and its compilation is informally explained in Section 6.2. We omit polymorphism here because it adds considerable complications that would distract us from the key ideas of the compilation scheme. Furthermore, our formalization does not include most optimizations.

The *target calculus* λ_r is a standard λ-calculus that supports extensible records, which can be regarded as a functional subset of JavaScript.

In summary, the formalization includes the key idea of compiling merges to type-indexed records, and the following improvements:

- The use of a new coercive style that avoids modeling coercions as function terms.
- Avoiding coercions for record projections, which were needed by Fan et al. [2022].

Technical results include proofs of type safety, as well as several interesting properties about our translation of types into record labels. All proofs are mechanically checked using the Coq proof assistant and are available in the supplementary materials.

5.1 Target Calculus with Extensible Records

As we have emphasized, our source language CP only allows disjoint traits in trait composition. Correspondingly, our source calculus λ_i^+ enforces the disjointness constraint on merges and does not accept records with overlapping fields. In contrast, the main characteristic of our *target* calculus λ_r is that it allows duplicate labels in records. When labels conflict, overriding happens, like the design of scoped labels by Leijen [2005]. But this overriding does not affect type safety (with the existence of subtyping) or the coherence of the elaboration semantics. This is because we only need to consider the terms that are generated by the elaboration from our source calculus λ_i^+. Since labels are computed from the corresponding source types of the fields, the type system of λ_r can require that duplicate labels in one record must be associated with fields of *equivalent* types. Besides, these fields are semantically equivalent because they *originate* from the same terms.

For instance, $1 \,,\, 2$ and even $1 \,,\, 1$ are forbidden in λ_i^+ (and our source language CP). Consequently, the elaborated terms in λ_r cannot have conflicting fields like $\{\texttt{int} \Mapsto 1;\ \texttt{int} \Mapsto 2\}$. However, it is possible, as part of evaluation, that harmless forms of duplicate fields arise, leading to duplicate fields where the values are the same, such as $\{\texttt{int} \Mapsto 1;\ \texttt{int} \Mapsto 1\}$. We will discuss this harmless duplication and the coherence of the elaboration semantics in Section 5.3 and Section 5.4 after presenting both calculi and the elaboration rules.

Syntax. We use the integer type as a representative of base types. $\mathbb{Z}$ denotes the integer type, and n represents any integer literal. The meta-variable ρ stands for record types, including the empty record type $\{\,\}$. The type ρ extended by a field of type $\mathcal{A}$ with label ℓ is written as $\{\ell \Mapsto \mathcal{A} \mid \rho\}$. For example, $\{\ell_1 \Mapsto \mathcal{A} \mid \{\ell_2 \Mapsto \mathcal{B} \mid \{\,\}\}\}$ is a record type with two fields and is abbreviated as

$\boxed{\epsilon \longrightarrow \epsilon'}$ *(Small-step semantics)*

$$\textsc{TStep-Proj}\ \frac{\epsilon \longrightarrow \epsilon'}{\epsilon.\ell \longrightarrow \epsilon'.\ell} \qquad \textsc{TStep-AppL}\ \frac{\epsilon_1 \longrightarrow \epsilon_1'}{\epsilon_1\,\epsilon_2 \longrightarrow \epsilon_1'\,\epsilon_2} \qquad \textsc{TStep-AppR}\ \frac{\epsilon \longrightarrow \epsilon'}{v\,\epsilon \longrightarrow v\,\epsilon'} \qquad \textsc{TStep-ConcatL}\ \frac{\epsilon_1 \longrightarrow \epsilon_1'}{\epsilon_1 +\!\!+\, \epsilon_2 \longrightarrow \epsilon_1' +\!\!+\, \epsilon_2} \qquad \textsc{TStep-ConcatR}\ \frac{\epsilon \longrightarrow \epsilon'}{v +\!\!+\, \epsilon \longrightarrow v +\!\!+\, \epsilon'}$$

$$\textsc{TStep-ProjRcd}\ \frac{\textbf{lookup}\ \ell\ v_1 \Rightarrow v_2}{v_1.\ell \longrightarrow v_2} \qquad \textsc{TStep-Rcd}\ \frac{\epsilon \longrightarrow \epsilon'}{\{\,\overline{\ell_i \Mapsto v_i}^{\,i};\ \ell \Mapsto \epsilon;\ \overline{\ell_j \Mapsto \epsilon_j}^{\,j}\,\} \longrightarrow \{\,\overline{\ell_i \Mapsto v_i}^{\,i};\ \ell \Mapsto \epsilon';\ \overline{\ell_j \Mapsto \epsilon_j}^{\,j}\,\}}$$

$$\textsc{TStep-Concat}\ \frac{}{\{\,\overline{\ell_i \Mapsto v_i}^{\,i}\,\} +\!\!+ \{\,\overline{\ell_j \Mapsto v_j}^{\,j}\,\} \longrightarrow \{\,\overline{\ell_i \Mapsto v_i}^{\,i};\ \overline{\ell \Mapsto v_j}^{\,j}\,\}} \qquad \textsc{TStep-AppAbs}\ \frac{}{(\lambda x.\,\epsilon)\ v \longrightarrow \epsilon[x \mapsto v]}$$

$\boxed{\textbf{lookup}\ \ell\ v_1 \Rightarrow v_2}$ *(Label lookup on records)*

$$\textbf{lookup}\ \ell\ \{\ell_1 \Mapsto v_1;\ \ldots;\ \ell_n \Mapsto v_n\} \Rightarrow v_k \qquad \text{if } \ell_k = \ell \text{ and } \forall j \in 1..k{-}1, \ell_j \neq \ell$$

$\boxed{\textbf{lookup}\ \ell\ \rho \Rightarrow \mathcal{B}}$ *(Label lookup on record types)*

$$\begin{array}{ll}
\textbf{lookup}\ \ell\ \{\ell \Mapsto \mathcal{A} \mid \rho\} \Rightarrow \mathcal{A} & \\
\textbf{lookup}\ \ell_1\ \{\ell_2 \Mapsto \mathcal{A} \mid \rho\} \Rightarrow \mathcal{B} & \text{if } \ell_1 \neq \ell_2 \text{ and } \textbf{lookup}\ \ell_1\ \rho \Rightarrow \mathcal{B} \\
\textbf{lookup}\ \ell_1\ \{\ell_2 \Mapsto \mathcal{A} \mid \rho\} \not\Rightarrow & \text{if } \ell_1 \neq \ell_2 \text{ and } \textbf{lookup}\ \ell_1\ \rho \not\Rightarrow \\
\textbf{lookup}\ \ell\ \{\,\} \not\Rightarrow &
\end{array}$$

Fig. 8. Dynamic semantics and meta-functions for λ_r.

$\{\ell_1 \Mapsto \mathcal{A};\ \ell_2 \Mapsto \mathcal{B}\}$. In general, abbreviations $\{\ell_1 \Mapsto \mathcal{A}_1;\ \ldots;\ \ell_n \Mapsto \mathcal{A}_n\}$ represents a multi-field record type, and $\{\ell_1 \Mapsto \mathcal{A}_1;\ \ldots;\ \ell_n \Mapsto \mathcal{A}_n \mid \rho\}$ is the record type ρ being extended by n fields. At the term level, records can be concatenated using $+\!\!+$, and $\epsilon.\ell$ extracts the first ℓ field from ϵ. The full syntax of λ_r is as follows:[5]

$$\begin{array}{lrcl}
\text{Types} & \mathcal{A}, \mathcal{B}, C & ::= & \mathbb{Z} \mid \mathcal{A} \to \mathcal{B} \mid \rho \\
\text{Record types} & \rho & ::= & \{\,\} \mid \{\ell \Mapsto \mathcal{A} \mid \rho\} \\
\text{Expressions} & \epsilon & ::= & n \mid x \mid \lambda x.\,\epsilon \mid \epsilon_1\,\epsilon_2 \mid \{\ell_1 \Mapsto \epsilon_1;\ \ldots;\ \ell_n \Mapsto \epsilon_n\} \mid \epsilon.\ell \mid \epsilon_1 +\!\!+\, \epsilon_2 \\
\text{Values} & v & ::= & n \mid \lambda x.\,\epsilon \mid \{\ell_1 \Mapsto v_1;\ \ldots;\ \ell_n \Mapsto v_n\} \\
\text{Typing contexts} & \Delta & ::= & \cdot \mid \Delta, x : \mathcal{A}
\end{array}$$

Small-step semantics. The dynamic semantics of target expressions is defined at the top of Fig. 8. For conciseness, we also use a list comprehension representation $\{\,\overline{\ell_i \Mapsto \epsilon_i}^{\,i}\,\}$ for multi-field records. The evaluation is call-by-value, and record fields are eagerly evaluated. To concatenate two records, they have to be fully reduced to values and then merged in rule TStep-Concat. For example, $\{\ell \Mapsto 1 + 1\} +\!\!+\, \epsilon$ evaluates to $\{\ell \Mapsto 2;\ \ell_1 \Mapsto v_1;\ \ldots;\ \ell_n \Mapsto v_n\}$, assuming that ϵ evaluates to $\{\ell_1 \Mapsto v_1;\ \ldots;\ \ell_n \Mapsto v_n\}$. Rule TStep-ProjRcd uses the lookup function ($\textbf{lookup}\ \ell\ v_1 \Rightarrow v_2$) defined in the middle of Fig. 8 to extract the first field with a matched label.

Type-level lookup. Besides the value-level lookup function, we define a meta-function on record types at the bottom of Fig. 8 to reflect the behavior of field selection. It finds the first field type that

[5]In our Coq formalization, the bottom type and fixpoint expressions are also formalized in both source and target calculi. We omit them in the paper to better align with λ_i^+ [Bi et al. 2018], which does not support these features.

Type equivalence $\qquad \mathcal{A} \approx \mathcal{B} \triangleq \mathcal{A} \subseteq \mathcal{B} \wedge \mathcal{B} \subseteq \mathcal{A}$

$\boxed{\mathcal{A} \subseteq \mathcal{B}}$ *(Width subtyping)*

$$\frac{}{\mathcal{A} \subseteq \mathcal{A}}\ \text{RTS-Refl} \qquad \frac{\mathcal{A} \approx \mathcal{A}' \qquad \mathcal{B} \approx \mathcal{B}'}{\mathcal{A} \to \mathcal{B} \subseteq \mathcal{A}' \to \mathcal{B}'}\ \text{RTS-Arrow} \qquad \frac{\forall \ell\ C,\ \text{if } \textbf{lookup}\ \ell\ \rho_2 \Rightarrow C \text{ then } \textbf{lookup}\ \ell\ \rho_1 \approx C}{\rho_1 \subseteq \rho_2}\ \text{RTS-Rcd}$$

$\boxed{\textbf{lookup}\ \ell\ \rho \approx \mathcal{A}}$, $\boxed{\textbf{lookup}\ \ell\ \rho \sim \mathcal{A}}$ *(Abbreviations for lookup)*

$$\begin{array}{lcl} \textbf{lookup}\ \ell\ \rho \approx \mathcal{A} & \triangleq & \exists \mathcal{A}',\ \textbf{lookup}\ \ell\ \rho \Rightarrow \mathcal{A}' \wedge \mathcal{A}' \approx \mathcal{A} \\ \textbf{lookup}\ \ell\ \rho \sim \mathcal{A} & \triangleq & \textbf{lookup}\ \ell\ \rho \approx \mathcal{A} \vee \textbf{lookup}\ \ell\ \rho \not\Rightarrow \end{array}$$

$\boxed{\vdash \Delta}$ *(Context well-formedness)*

$$\frac{}{\vdash \cdot}\ \text{WFC-Nil} \qquad \frac{\vdash \mathcal{A} \qquad \vdash \Delta}{\vdash \Delta, x : \mathcal{A}}\ \text{WFC-Cons}$$

$\boxed{\vdash \mathcal{A}}$ *(Type well-formedness)*

$$\frac{}{\vdash \{\}}\ \text{WF-Nil} \qquad \frac{}{\vdash \mathbb{Z}}\ \text{WF-Int} \qquad \frac{\vdash \mathcal{A} \qquad \vdash \mathcal{B}}{\vdash \mathcal{A} \to \mathcal{B}}\ \text{WF-Arrow} \qquad \frac{\vdash \mathcal{A} \qquad \vdash \rho \qquad \textbf{lookup}\ \ell\ \rho \sim \mathcal{A}}{\vdash \{\ell \Mapsto \mathcal{A} \mid \rho\}}\ \text{WF-Rcd}$$

Fig. 9. Width subtyping, type equivalence, and well-formedness in λ_r.

matches the given label, just like the value-level one. We use $\textbf{lookup}\ \ell\ \rho \not\Rightarrow$ to represent the case where no field in ρ matches ℓ.

Width subtyping. We define a form of width subtyping for record types at the top of Fig. 9, while depth subtyping is not supported in λ_r. Intuitively, $\rho_1 \subseteq \rho_2$ holds if, for any projection that can be performed on a term of ρ_2, it can also be performed on any term of ρ_1, and their results have equivalent types. The subtyping relation will be used after we introduce our source calculus and its elaboration semantics in the next subsection. In the metatheory proofs, we will need to relate record expressions to parts of their types, like $\{\ell = 1;\ \ell' = \textbf{true}\}$ to $\{\ell \Mapsto \mathbb{Z}\}$. The relation between types and their parts is characterized by width subtyping.

Equivalence of target types. An equivalence relation $\approx$ is derived from width subtyping to allow permutation of record fields. $\textbf{lookup}\ \ell\ \rho \approx \mathcal{A}$ is an abbreviation for the case where looking up ℓ in ρ produces a type equivalent to $\mathcal{A}$. A similar abbreviation $\textbf{lookup}\ \ell\ \rho \sim \mathcal{A}$ additionally includes the case where ℓ is absent in ρ. An important property of equivalent types is that they preserve the results of lookup:

Lemma 5.1 (Lookup on equivalent types). *Given $\rho_1 \approx \rho_2$:*

- *If* $\textbf{lookup}\ \ell\ \rho_1 \Rightarrow C$ *then* $\textbf{lookup}\ \ell\ \rho_2 \approx C$.
- *If* $\textbf{lookup}\ \ell\ \rho_1 \not\Rightarrow$ *then* $\textbf{lookup}\ \ell\ \rho_2 \not\Rightarrow$.

Type well-formedness. Well-formed types are defined at the bottom of Fig. 9. Record type extension must be consistent: duplicate labels must be associated with equivalent field types. Specifically,

$\boxed{\rho_1 \uplus \rho_2 \Rightarrow \rho_3}$ *(Record type concatenation)*

$$\text{CT-NIL}\quad \frac{}{\{\,\} \uplus \rho \Rightarrow \rho} \qquad \text{CT-RCD}\quad \frac{\textbf{lookup}\ \ell\ \rho_2 \sim \mathcal{A} \qquad \rho_1 \uplus \rho_2 \Rightarrow \rho_3}{\{\ell \Mapsto \mathcal{A} \mid \rho_1\} \uplus \rho_2 \Rightarrow \{\ell \Mapsto \mathcal{A} \mid \rho_3\}}$$

$\boxed{\Delta \vdash \epsilon : \mathcal{A}}$ *(Typing)*

$$\text{TYP-INT}\quad \frac{\vdash \Delta}{\Delta \vdash n : \mathbb{Z}} \qquad \text{TYP-ABS}\quad \frac{\Delta, x : \mathcal{A} \vdash \epsilon : \mathcal{B}}{\Delta \vdash \lambda x.\, \epsilon : \mathcal{A} \to \mathcal{B}} \qquad \text{TYP-APP}\quad \frac{\Delta \vdash \epsilon_1 : \mathcal{A} \to \mathcal{B} \qquad \Delta \vdash \epsilon_2 : \mathcal{A}' \qquad \mathcal{A} \approx \mathcal{A}'}{\Delta \vdash \epsilon_1\, \epsilon_2 : \mathcal{B}}$$

$$\text{TYP-RCDNIL}\quad \frac{\vdash \Delta}{\Delta \vdash \{\,\} : \{\,\}} \qquad \text{TYP-RCDCONS}\quad \frac{\Delta \vdash \epsilon : \mathcal{A} \qquad \Delta \vdash \{\ell_1 \Mapsto \epsilon_1;\ \ldots;\ \ell_n \Mapsto \epsilon_n\} : \rho \qquad \textbf{lookup}\ \ell\ \rho \sim \mathcal{A}}{\Delta \vdash \{\ell \Mapsto \epsilon;\ \ell_1 \Mapsto \epsilon_1;\ \ldots;\ \ell_n \Mapsto \epsilon_n\} : \{\ell \Mapsto \mathcal{A} \mid \rho\}}$$

$$\text{TYP-VAR}\quad \frac{\vdash \Delta \qquad x : \mathcal{A} \in \Delta}{\Delta \vdash x : \mathcal{A}} \qquad \text{TYP-RCDPROJ}\quad \frac{\Delta \vdash \epsilon : \rho \qquad \textbf{lookup}\ \ell\ \rho \Rightarrow \mathcal{B}}{\Delta \vdash \epsilon.\ell : \mathcal{B}} \qquad \text{TYP-RCDMERGE}\quad \frac{\rho_1 \uplus \rho_2 \Rightarrow \rho_3 \qquad \Delta \vdash \epsilon_1 : \rho_1 \qquad \Delta \vdash \epsilon_2 : \rho_2}{\Delta \vdash \epsilon_1 +\!\!+\, \epsilon_2 : \rho_3}$$

Fig. 10. Typing of λ_r.

as shown by rule WF-RCD, to safely extend type ρ by a new field of label ℓ, either the old field type in ρ is equivalent to the new field type, or ρ lacks label ℓ. This is also enforced in the typing rule TYP-RCDCONS.

As we will explain later, every type in the source language, including an intersection type, is translated into a record type in the target language. All the record labels are generated from source types in the translation process, where disjoint source types are converted to distinct labels. Although overlapping is forbidden in merges, overlapping is *not* forbidden in intersection types. For example, $1 \,,\, 2$ is forbidden but $\mathbb{Z} \,\&\, \mathbb{Z}$ is a valid type in λ_i^+. Therefore, it is natural for corresponding record types to contain duplicate labels. The properties of the source calculus also ensure that the translated types are well-formed. With the well-formedness restriction, permuting any fields in a record type does not affect type safety.

Typing. The typing rules of target expressions are presented in Fig. 10. A set of auxiliary rules is defined to concatenate two record types ($\rho_1 \uplus \rho_2 \Rightarrow \rho_3$). The premise of rule CT-RCD guarantees the well-formedness of the result type. Given the types of two record expressions, the concatenation of the two record types directly reveals the shape of the result of concatenating the two expressions.

In our type system, there is no subsumption rule or a rule that allows conversion between $\approx$-equivalent types. Every expression under the given typing context has a unique type. That is, from a record type, it is straightforward to tell the shape of its value: how many fields it has, what the labels are, and how the fields are arranged. On the other hand, in rule TYP-APP, the requirement on argument type is relaxed to $\approx$-equivalence.

Type soundness. The λ_r calculus is proven to be type-sound via progress and type preservation. However, we should emphasize that type preservation (and the substitution lemma) is proven with respect to $\approx$-equivalence.

THEOREM 5.2 (PROGRESS). *If $\cdot \vdash \epsilon : \mathcal{A}$, then either ϵ is a value or $\exists \epsilon'$, $\epsilon \longrightarrow \epsilon'$.*

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

$$\text{TL-Top}\quad \frac{}{\rceil \top \lceil} \qquad \text{TL-And}\quad \frac{\rceil A \lceil \quad \rceil B \lceil}{\rceil A \,\&\, B \lceil} \qquad \text{TL-Arrow}\quad \frac{\rceil B \lceil}{\rceil A \to B \lceil} \qquad \text{TL-Rcd}\quad \frac{\rceil B \lceil}{\rceil \{\ell : B\} \lceil}$$

$\boxed{A * B}$ *(Type disjointness)*

$$\text{D-Symm}\quad \frac{B * A}{A * B} \qquad \text{D-TopL}\quad \frac{\rceil A \lceil}{A * B} \qquad \text{D-AndL}\quad \frac{A_1 * B \quad A_2 * B}{A_1 \,\&\, A_2 * B} \qquad \text{D-IntArrow}\quad \frac{}{\mathbb{Z} * A_1 \to A_2} \qquad \text{D-IntRcd}\quad \frac{}{\mathbb{Z} * \{\ell : A\}}$$

$$\text{D-ArrowArrow}\quad \frac{A_2 * B_2}{A_1 \to A_2 * B_1 \to B_2} \qquad \text{D-RcdEq}\quad \frac{A * B}{\{\ell : A\} * \{\ell : B\}} \qquad \text{D-RcdNeq}\quad \frac{\ell_1 \neq \ell_2}{\{\ell_1 : A\} * \{\ell_2 : B\}} \qquad \text{D-ArrowRcd}\quad \frac{}{A_1 \to A_2 * \{\ell : A\}}$$

Fig. 11. Top-like types and type disjointness in λ_i^+.

Lemma 5.3 (Substitution preserves typing). *If $\Delta, x : \mathcal{A}, \Delta' \vdash \epsilon : \mathcal{B}$ and $\Delta \vdash \epsilon' : \mathcal{A}'$ and $\mathcal{A}' \approx \mathcal{A}$, then $\exists\, \mathcal{B}'$ such that $\Delta, \Delta' \vdash \epsilon[x \mapsto \epsilon'] : \mathcal{B}'$ and $\mathcal{B}' \approx \mathcal{B}$.*

Theorem 5.4 (Type preservation). *If $\cdot \vdash \epsilon : \mathcal{A}$ and $\epsilon \longrightarrow \epsilon'$, then $\exists \mathcal{A}', \cdot \vdash \epsilon' : \mathcal{A}'$ and $\mathcal{A}' \approx \mathcal{A}$.*

5.2 Source Calculus and Elaboration

The source calculus is a variant of λ_i^+ [Bi et al. 2018; Huang et al. 2021]. It includes type $\top$, the maximal element in subtyping, as well as its canonical value $\top$. Functions ($\lambda x.\, e : A \to B$) always have type annotations. $\{\ell = e\}$ stands for single-field records, which has type $\{\ell : A\}$ if e has type A. The full syntax of λ_i^+ is as follows:

$$\begin{array}{lrcl} \text{Types} & A, B, C & ::= & \top \mid \mathbb{Z} \mid A \to B \mid \{\ell : A\} \mid A \,\&\, B \\ \text{Expressions} & e & ::= & \top \mid n \mid x \mid \lambda x.\, e : A \to B \mid e_1\, e_2 \mid \{\ell = e\} \mid e.\ell \mid e_1 \,,\, e_2 \mid e : A \end{array}$$

Merge operator and disjoint intersection types. The symmetric merge operator (,) is like record concatenation, with which we can construct multi-field records from single-field records. However, it is not restricted to records: as long as two expressions have *disjoint* types (i.e. they are thought to be compatible), $e_1 \,,\, e_2$ is allowed, containing the information of both expressions. Assuming that e_1 and e_2 have type A and B respectively, the whole merge has intersection type $A \,\&\, B$.

$$\begin{array}{rcl} \{\ell_1 : A_1;\ \ldots;\ \ell_n : A_n\} & \triangleq & \{\ell_1 : A_1\} \,\&\, \ldots \,\&\, \{\ell_n : A_n\} \\ \{\ell_1 = e_1;\ \ldots;\ \ell_n = e_n\} & \triangleq & \{\ell_1 = e_1\} \,,\, \ldots \,,\, \{\ell_n = e_n\} \end{array}$$

Top-like types and disjointness. Fig. 11 defines two relations. They follow the specifications in previous work on disjoint intersection types [Huang et al. 2021], which are defined in terms of coercive subtyping ($\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$) in Fig. 15. Since we only need to consider the relation on types, here we use $A <: B$ to represent the subtyping relation, ignoring the terms ϵ_1 and ϵ_2.

Theorem 5.5 (Coercion-erased subtyping). *$A <: B$ if and only if $\forall \epsilon_1, \exists \epsilon_2, \epsilon_1 : A <: B \rightsquigarrow \epsilon_2$.*

At the top of Fig. 11 is the algorithmic definition of top-like types ($\rceil A \lceil$). It characterizes types that are equivalent to $\top$, including any function type with a top-like return type, and any record type with a top-like field type. These types are thought to be vacuous and treated in a unified way.

Theorem 5.6 (Top-like types respect the specification). *$\rceil A \lceil$ if and only if $\top <: A$.*

$$
\begin{array}{lrcl}
\text{Type indices} & T & ::= & \overline{\mathbb{Z}} \mid \overline{Ts \to Ts} \mid \overline{\{\ell : Ts\}} \\
\text{List of type indices} & Ts & ::= & [\,T_1, \ldots, T_n\,]
\end{array}
$$

$\boxed{|A| = Ts}$ *(Translation to type indices)*

$$
\begin{array}{rcll}
|A| & = & [\ ] & \text{if } \rceil A \lceil \\
|\mathbb{Z}| & = & [\overline{\mathbb{Z}}] & \\
|A \to B| & = & [\overline{|A| \to |B|}] & \text{if not } \rceil B \lceil \\
|\{\ell : A\}| & = & [\overline{\{\ell : |A|\}}] & \text{if not } \rceil A \lceil \\
|A \,\&\, B| & = & \textbf{dedup}\ (\textbf{merge}\ |A|\ |B|) &
\end{array}
$$

$\boxed{\|A\| = \rho}$ *(Translation to target types)*

$$
\begin{array}{rcll}
\|A\| & = & \{\ \} & \text{if } \rceil A \lceil \\
\|\mathbb{Z}\| & = & \{\overline{\mathbb{Z}} \Mapsto \mathbb{Z}\} & \\
\|A \to B\| & = & \{\overline{|A| \to |B|} \Mapsto \|A\| \to \|B\|\} & \text{if not } \rceil B \lceil \\
\|\{\ell : A\}\| & = & \{\overline{\{\ell : |A|\}} \Mapsto \|A\|\} & \text{if not } \rceil A \lceil \\
\|A \,\&\, B\| & = & \rho & \text{if } \|A\| \uplus \|B\| \Rightarrow \rho
\end{array}
$$

Fig. 12. Translation functions for types in λ_i^+.

At the bottom of Fig. 11 is the algorithmic definition of disjointness. We say two types are disjoint ($A * B$) if they do not overlap on any meaningful types; or, any common supertypes they share are top-like. Irrelevant types are considered disjoint, such as integer and function types, or records with different labels. Function types are disjoint if and only if their return types are disjoint. Two record types with the same label are disjoint if and only if their field types are disjoint.

Theorem 5.7 (Type disjointness respects the specification). *$A * B$ if and only if $\forall C$, if $A <: C$ and $B <: C$ then $\rceil C \lceil$.*

Type indices and translation functions. Merges in λ_i^+ are elaborated into records in λ_r. Each component is tagged by a label, which we call a *type index*. Type indices are computed from the component types of an intersection. Defined in Fig. 12, the translation function $|\cdot|$ maps a type to a list of type indices Ts. For types that are neither a top-like nor an intersection type, the result is a singleton list. Values of top-like types are thought to contain no information, so these types are omitted in translation, i.e. they are converted into an empty list [ ]. These lists are merged in the case of intersection types: **merge** is a merge sort, taking two sorted lists and producing a merged sorted list. Then we remove duplicates from the result list using **dedup**. For example, $\mathbb{Z} \,\&\, (\mathbb{Z} \to \mathbb{Z}) \,\&\, \mathbb{Z} \,\&\, (\top \to \top)$ is translated to $[\overline{\mathbb{Z}}, \overline{\overline{\mathbb{Z}} \to \overline{\mathbb{Z}}}]$. The list only contains the type indices for the first two elements of the intersection type because the third element is a duplicate of the first one and the last element is a top-like type. We use an injective function to map each type index to a unique string in Coq, and we use the alphabetical order of their corresponding strings to sort type indices.

Lemma 5.8 (Translation). *The mapping from type indices to strings has the following properties:*

- *If $|A_1 \to B_1| = |A_2 \to B_2|$ then $|A_1| = |A_2|$ and $|B_1| = |B_2|$, given that $A_1 \to B_1$ and $A_2 \to B_2$ are not top-like.*
- *If $|\{\ell_1 : A_1\}| = |\{\ell_2 : A_2\}|$ then $\ell_1 = \ell_2$ and $|A_1| = |A_2|$, given that $\{\ell_1 : A_1\}$ and $\{\ell_2 : A_2\}$ are not top-like.*

To the right of the type-index translation function, there is another function $\|\cdot\|$ that maps source types to target types. It uses the record type concatenation defined in Fig. 10. The function is based on the design of elaboration, which we will introduce later (presented in Fig. 14). It reflects the type of the elaborated target term. The result of translation is always a record type: all top-like types are converted to the empty record type; converting an intersection type is concatenating

$$\text{Type equivalence} \qquad A \risingdotseq B \quad \triangleq \quad A \sqsubseteq B \wedge B \sqsubseteq A$$

$\boxed{A \sqsubseteq B}$ *(Width subtyping)*

$$\textsc{RS-Int}\ \frac{}{\mathbb{Z} \sqsubseteq \mathbb{Z}} \qquad \textsc{RS-Top}\ \frac{\rceil B \lceil}{A \sqsubseteq B} \qquad \textsc{RS-Arrow}\ \frac{A_1 \risingdotseq B_1 \quad A_2 \risingdotseq B_2}{A_1 \to A_2 \sqsubseteq B_1 \to B_2} \qquad \textsc{RS-Rcd}\ \frac{A \risingdotseq B}{\{\ell : A\} \sqsubseteq \{\ell : B\}} \qquad \textsc{RS-AndL1}\ \frac{A_1 \sqsubseteq A_3}{A_1 \,\&\, A_2 \sqsubseteq A_3}$$

$$\textsc{RS-AndL2}\ \frac{A_2 \sqsubseteq A_3}{A_1 \,\&\, A_2 \sqsubseteq A_3} \qquad \textsc{RS-AndR}\ \frac{A_1 \sqsubseteq A_2 \qquad A_1 \sqsubseteq A_3}{A_1 \sqsubseteq A_2 \,\&\, A_3}$$

Fig. 13. Width subtyping in λ_i^+.

their counterparts. For the remaining types, the translation is a record type tagged by the type index associated with the type itself. Only when two field types are $\approx$-equivalent, they can have the same type index. While our typing rules use the type-index translation function $|\cdot|$, the type translation function $\|\cdot\|$ only serves the purpose of proving metatheory properties.

Equivalence of source types. Corresponding to the $\approx$-equivalence on target types, $\risingdotseq$ defines an equivalence relation on source types. Likewise, it is derived from a preorder ($A \sqsubseteq B$), which is the width subtyping in the source calculus. Note that it is not the subtyping used in the type system, but rather an auxiliary relation defined to better characterize the invariant of the type index translation. As defined in Fig. 13, this preorder relation is stricter than the coercive subtyping used in our source type system (presented in Fig. 15). An intersection type can be intuitively understood as a set of distinct types. For example, the intersection type **Bool** & **Char** & (**Int** $\to$ **Int**) represents a set of three distinct elements: **Bool**, **Char**, and **Int** $\to$ **Int**. Its width subtype must contain all these three elements. Generally speaking, all component types in an intersection must be present in its width subtype, excluding duplicates and top-like types. The $\risingdotseq$-equivalence groups types that map to the same type index.

Lemma 5.9 (Equivalent types). *Some properties of the $\risingdotseq$-equivalence can be derived from properties of width subtyping:*

- *If* $\textbf{lookup}\ \ell\ \|A\| \Rightarrow C_1$ *and* $\textbf{lookup}\ \ell\ \|B\| \Rightarrow C_2$ *then* $C_1 \subseteq C_2$. *Thus, by symmetry, if* $\textbf{lookup}\ \ell\ \|A\| \Rightarrow C_1$ *and* $\textbf{lookup}\ \ell\ \|B\| \Rightarrow C_2$ *then* $C_1 \approx C_2$.
- $A \sqsubseteq B$ *if and only if all components of* $|B|$ *can be found in* $|A|$. *Thus, by symmetry,* $A \risingdotseq B$ *if and only if* $|A| = |B|$.
- *If* $A \sqsubseteq B$ *then* $\|A\| \subseteq \|B\|$. *Thus, by symmetry, if* $A \risingdotseq B$ *then* $\|A\| \approx \|B\|$.

The first one is a strong result about type translation: in a translated type, the type of a record field can be determined by its associated label. Hence, for any two translated types $\|A\|$ and $\|B\|$, looking up the same label ℓ will lead to equivalent types C_1 and C_1. For example, looking up an integer label $\overline{\mathbb{Z}}$ should always return an integer type $\mathbb{Z}$. With the above properties, we can prove that all translated types are well-formed.

Lemma 5.10 (Well-formedness of translated types). $\forall A, \vdash \|A\|$.

Type-directed elaboration. Defined in Fig. 14, $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow \epsilon$ relates a source expression e to a source type A under the typing context Γ and the typing mode $\Leftrightarrow$, and the typing derivation

Typing contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Typing modes	$\Leftrightarrow ::= \Leftarrow \mid \Rightarrow$

$\boxed{\Gamma \vdash e \Leftrightarrow A \rightsquigarrow \epsilon}$ *(Type-directed elaboration)*

$$\textsc{Ela-Top}\quad \frac{}{\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \{\,\}}$$

$$\textsc{Ela-TopAbs}\quad \frac{\rceil B \lceil}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B \rightsquigarrow \{\,\}}$$

$$\textsc{Ela-TopRcd}\quad \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon \qquad \rceil A \lceil}{\Gamma \vdash \{\ell = e\} \Rightarrow \{\ell : A\} \rightsquigarrow \{\,\}}$$

$$\textsc{Ela-Int}\quad \frac{}{\Gamma \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \{\overline{\overline{\mathbb{Z}}} \Mapsto n\}}$$

$$\textsc{Ela-Var}\quad \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \rightsquigarrow x}$$

$$\textsc{Ela-Abs}\quad \frac{\neg\rceil B \lceil \qquad \Gamma, x : A \vdash e \Leftarrow B \rightsquigarrow \epsilon}{\Gamma \vdash \lambda x.\, e : A \to B \Rightarrow A \to B \rightsquigarrow \{|A \to B| \Mapsto \lambda x.\, \epsilon\}}$$

$$\textsc{Ela-Sub}\quad \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \epsilon_1 : A <: B \rightsquigarrow \epsilon_2}{\Gamma \vdash e \Leftarrow B \rightsquigarrow \epsilon_2}$$

$$\textsc{Ela-App}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \Gamma \vdash e_2 \Rightarrow B \rightsquigarrow \epsilon_2 \qquad \epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \epsilon_3 : C}{\Gamma \vdash e_1\, e_2 \Rightarrow C \rightsquigarrow \epsilon_3}$$

$$\textsc{Ela-Rcd}\quad \frac{\neg\rceil A \lceil \qquad \Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon}{\Gamma \vdash \{\ell = e\} \Rightarrow \{\ell : A\} \rightsquigarrow \{|\{\ell : A\}| \Mapsto \epsilon\}}$$

$$\textsc{Ela-Proj}\quad \frac{\Gamma \vdash e \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_2 : B}{\Gamma \vdash e.\ell \Rightarrow B \rightsquigarrow \epsilon_2}$$

$$\textsc{Ela-Merge}\quad \frac{\Gamma \vdash e_1 \Rightarrow A \rightsquigarrow \epsilon_1 \qquad \Gamma \vdash e_2 \Rightarrow B \rightsquigarrow \epsilon_2 \qquad A * B}{\Gamma \vdash e_1 \,,\!,\, e_2 \Rightarrow A \,\&\, B \rightsquigarrow \epsilon_1 \mathbin{+\!\!+} \epsilon_2}$$

$$\textsc{Ela-Anno}\quad \frac{\Gamma \vdash e \Leftarrow A \rightsquigarrow \epsilon}{\Gamma \vdash e : A \Rightarrow A \rightsquigarrow \epsilon}$$

$\boxed{\epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \epsilon_3 : C}$ *(Distributive application)*

$$\textsc{A-Top}\quad \frac{\rceil A \lceil}{\epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \{\,\} : \top}$$

$$\textsc{A-Arrow}\quad \frac{\neg\rceil B \lceil \qquad \epsilon_2 : C <: A \rightsquigarrow \epsilon_3}{\epsilon_1 : A \to B \bullet \epsilon_2 : C \rightsquigarrow (\epsilon_1.|A \to B|)\, \epsilon_3 : B}$$

$$\textsc{A-And}\quad \frac{\neg\rceil A \,\&\, B \lceil \qquad \epsilon_1 : A \bullet \epsilon_2 : C \rightsquigarrow \epsilon_3 : A' \qquad \epsilon_1 : B \bullet \epsilon_2 : C \rightsquigarrow \epsilon_4 : B'}{\epsilon_1 : A \,\&\, B \bullet \epsilon_2 : C \rightsquigarrow \epsilon_3 \mathbin{+\!\!+} \epsilon_4 : A' \,\&\, B'}$$

$\boxed{\epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_3 : C}$ *(Distributive projection)*

$$\textsc{P-Top}\quad \frac{\rceil A \lceil}{\epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \{\,\} : \top}$$

$$\textsc{P-RcdEq}\quad \frac{\neg\rceil A \lceil}{\epsilon : \{\ell : A\} \bullet \{\ell\} \rightsquigarrow \epsilon.|\{\ell : A\}| : A}$$

$$\textsc{P-RcdNeq}\quad \frac{\neg\rceil A \lceil \qquad \ell_1 \neq \ell_2}{\epsilon : \{\ell_1 : A\} \bullet \{\ell_2\} \rightsquigarrow \{\,\} : \top}$$

$$\textsc{P-And}\quad \frac{\neg\rceil A \,\&\, B \lceil \qquad \epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_2 : A' \qquad \epsilon_1 : B \bullet \{\ell\} \rightsquigarrow \epsilon_3 : B'}{\epsilon_1 : A \,\&\, B \bullet \{\ell\} \rightsquigarrow \epsilon_2 \mathbin{+\!\!+} \epsilon_3 : A' \,\&\, B'}$$

Fig. 14. Type-directed elaboration of λ_i^{+}.

$$\text{Ordinary types} \qquad A^\circ, B^\circ, C^\circ ::= \top \mid \mathbb{Z} \mid A \to B^\circ \mid \{\ell : A^\circ\}$$

$\boxed{\epsilon_1 : A <: B \rightsquigarrow \epsilon_2}$ *(Coercive subtyping)*

$$\text{S-TOP}\ \frac{\rceil B^\circ \lceil}{\epsilon : A <: B^\circ \rightsquigarrow \{\,\}} \qquad \text{S-INT}\ \frac{}{\epsilon : \mathbb{Z} <: \mathbb{Z} \rightsquigarrow \{\overline{\mathbb{Z}} \Rightarrow \epsilon.\overline{\mathbb{Z}}\}}$$

$$\text{S-ARROW}\ \frac{\neg\rceil B_2^\circ \lceil \qquad x : B_1 <: A_1 \rightsquigarrow \epsilon_1 \qquad (\epsilon.|A_1 \to A_2|)\,\epsilon_1 : A_2 <: B_2^\circ \rightsquigarrow \epsilon_2}{\epsilon : A_1 \to A_2 <: B_1 \to B_2^\circ \rightsquigarrow \{|B_1 \to B_2^\circ| \Rightarrow \lambda x.\, \epsilon_2\}} \qquad \text{S-ANDL}\ \frac{\epsilon : A <: C^\circ \rightsquigarrow \epsilon'}{\epsilon : A \,\&\, B <: C^\circ \rightsquigarrow \epsilon'}$$

$$\text{S-RCD}\ \frac{\neg\rceil B^\circ \lceil \qquad \epsilon.|\{\ell : A\}| : A <: B^\circ \rightsquigarrow \epsilon_2}{\epsilon : \{\ell : A\} <: \{\ell : B^\circ\} \rightsquigarrow \{|\{\ell : B^\circ\}| \Rightarrow \epsilon_2\}} \qquad \text{S-ANDR}\ \frac{\epsilon : B <: C^\circ \rightsquigarrow \epsilon'}{\epsilon : A \,\&\, B <: C^\circ \rightsquigarrow \epsilon'}$$

$$\text{S-SPLIT}\ \frac{B_1 \lhd B \rhd B_2 \qquad \epsilon : A <: B_1 \rightsquigarrow \epsilon_1 \qquad \epsilon : A <: B_2 \rightsquigarrow \epsilon_2 \qquad \epsilon_1 : B_1 \rhd B \lhd \epsilon_2 : B_2 \rightsquigarrow \epsilon_3}{\epsilon : A <: B \rightsquigarrow \epsilon_3}$$

Fig. 15. Coercive subtyping in λ_i^+.

generates a target expression ϵ from e. The type system is *bidirectional* [Dunfield and Krishnaswami 2021; Pierce and Turner 2000]: under the inference mode ($\Rightarrow$), A is generated as an output; under the checking mode ($\Leftarrow$), A is given as an input. Given the typing context, every well-typed e has a unique inferred type; all the types that e can be checked against are supertypes of this inferred type.

Rule ELA-MERGE is the signature rule of calculi with *disjoint intersection types*. The disjointness restriction on types ($A * B$, defined in Fig. 11) prevents the overlapping of components in a merge. Thus, in a well-typed term like $e_1 ,, \ldots ,, e_n$, every subterm in the merge has disjoint types. Rule ELA-ANNO allows upcasting expressions to any supertypes. The subtyping relation is checked in rule ELA-SUB via the subtyping judgment $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$, which also coerces the target term ϵ_1 to ϵ_2. Rule ELA-APP relies on *distributive application*, which is defined in the middle of Fig. 14. It takes the function type A and the argument type B and, if A can be applied to B, produces the return type C. Distributive application additionally allows intersection types and top-like types (can be regarded as 0-ary intersections) to be applicable due to the distributivity of functions over intersections. For example, $(A_1 \to B_1) \,\&\, (A_2 \to B_2)$ can be applied to $A_1 \,\&\, A_2$ and produces $B_1 \,\&\, B_2$. Besides, $\epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \epsilon_3 : C$ uses ϵ_1 and ϵ_2 to generate the target term ϵ_3, reflecting the application in the target language. Similarly, rule ELA-PROJ relies on *distributive projection* to obtain the result type. Given a label ℓ, the relation $\epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_2 : B$ finds all field types in A that match ℓ and returns them as an intersection type B, if there is more than one matched field. Similarly, ϵ_2 is the target expression that extracts the corresponding fields in ϵ_1.

Rules ELA-TOP, ELA-TOPABS, and ELA-TOPRCD generate an empty record for top-like types, which is a counterpart of the canonical top value. For non-top-like types, rules ELA-INT, ELA-ABS, and ELA-RCD produces records with a single label translated from the type directly. Consequently, all elaborated terms are either reducible or are in a record form.

$\boxed{B \lhd A \rhd C}$ *(Type splitting)*

$$\textsc{Sp-And}\quad \frac{}{A \lhd A \& B \rhd B} \qquad \textsc{Sp-Arrow}\quad \frac{B_1 \lhd B \rhd B_2}{A \to B_1 \lhd A \to B \rhd A \to B_2} \qquad \textsc{Sp-Rcd}\quad \frac{B_1 \lhd B \rhd B_2}{\{\ell : B_1\} \lhd \{\ell : B\} \rhd \{\ell : B_2\}}$$

$\boxed{\epsilon_1 : A \rhd C \lhd \epsilon_2 : B \rightsquigarrow \epsilon}$ *(Coercive merging)*

$$\textsc{M-Top}\quad \frac{\rceil C \lceil \qquad A \lhd C \rhd B}{\epsilon_1 : A \rhd C \lhd \epsilon_2 : B \rightsquigarrow \{\}} \qquad \textsc{M-And}\quad \frac{\neg \rceil A \& B \lceil}{\epsilon_1 : A \rhd A \& B \lhd \epsilon_2 : B \rightsquigarrow \epsilon_1 +\!\!+\, \epsilon_2}$$

$$\textsc{M-Arrow}\quad \frac{\begin{array}{c}\neg \rceil B_1 \lceil \qquad \neg \rceil B_2 \lceil \\ (\epsilon_1.|A \to B_1|)\, x : B_1 \rhd B \lhd (\epsilon_2.|A \to B_2|)\, x : B_2 \rightsquigarrow \epsilon\end{array}}{\epsilon_1 : A \to B_1 \rhd A \to B \lhd \epsilon_2 : A \to B_2 \rightsquigarrow \{|A \to B| \Mapsto \lambda x.\, \epsilon\}}$$

$$\textsc{M-ArrowL}\quad \frac{\begin{array}{c}\neg \rceil B_1 \lceil \qquad \rceil B_2 \lceil \\ (\epsilon_1.|A \to B_1|)\, x : B_1 \rhd B \lhd \{\} : B_2 \rightsquigarrow \epsilon\end{array}}{\epsilon_1 : A \to B_1 \rhd A \to B \lhd \epsilon_2 : A \to B_2 \rightsquigarrow \{|A \to B| \Mapsto \lambda x.\, \epsilon\}}$$

$$\textsc{M-ArrowR}\quad \frac{\begin{array}{c}\rceil B_1 \lceil \qquad \neg \rceil B_2 \lceil \\ \{\} : B_1 \rhd B \lhd (\epsilon_2.|A \to B_2|)\, x : B_2 \rightsquigarrow \epsilon\end{array}}{\epsilon_1 : A \to B_1 \rhd A \to B \lhd \epsilon_2 : A \to B_2 \rightsquigarrow \{|A \to B| \Mapsto \lambda x.\, \epsilon\}}$$

$$\textsc{M-Rcd}\quad \frac{\begin{array}{c}\neg \rceil A_1 \lceil \qquad \neg \rceil A_2 \lceil \\ \epsilon_1.|\{\ell : A_1\}| : A_1 \rhd A \lhd \epsilon_2.|\{\ell : A_2\}| : A_2 \rightsquigarrow \epsilon\end{array}}{\epsilon_1 : \{\ell : A_1\} \rhd \{\ell : A\} \lhd \epsilon_2 : \{\ell : A_2\} \rightsquigarrow \{|\{\ell : A\}| \Mapsto \epsilon\}}$$

$$\textsc{M-RcdL}\quad \frac{\begin{array}{c}\neg \rceil A_1 \lceil \qquad \rceil A_2 \lceil \\ \epsilon_1.|\{\ell : A_1\}| : A_1 \rhd A \lhd \{\} : A_2 \rightsquigarrow \epsilon\end{array}}{\epsilon_1 : \{\ell : A_1\} \rhd \{\ell : A\} \lhd \epsilon_2 : \{\ell : A_2\} \rightsquigarrow \{|\{\ell : A\}| \Mapsto \epsilon\}}$$

$$\textsc{M-RcdR}\quad \frac{\begin{array}{c}\rceil A_1 \lceil \qquad \neg \rceil A_2 \lceil \\ \{\} : A_1 \rhd A \lhd \epsilon_2.|\{\ell : A_2\}| : A_2 \rightsquigarrow \epsilon\end{array}}{\epsilon_1 : \{\ell : A_1\} \rhd \{\ell : A\} \lhd \epsilon_2 : \{\ell : A_2\} \rightsquigarrow \{|\{\ell : A\}| \Mapsto \epsilon\}}$$

Fig. 16. Type splitting and coercive merging in λ_i^+.

Coercive subtyping. Defined in Fig. 15, $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$ takes a target expression ϵ_1 and two source types A and B and produces a target term ϵ_2. Intuitively, when ϵ_1 has type $\|A\|$, the generated ϵ_2 will have a type that is equivalent to $\|B\|$. The formal theorem will be given later (Theorem 5.11) when establishing type soundness. Besides producing the coerced target term, this relation also checks whether A is a subtype of B. For coercive subtyping, a more common form of the judgment is $A <: B \rightsquigarrow c$, where c is a coercion function in the target language with type $\|A\| \to \|B\|$. Instead of generating a coercion function, we directly transform the term. The main motivation behind our design is to generate more efficient terms: ϵ_2 can be understood as a simplified result of the application $c\ \epsilon_1$. By adopting this technique, we aim to skip some reduction steps, ultimately improving the performance of code that relies on coercions. This idea has been discussed in Section 4.1, and it is further optimized in Section 6.4.

To understand the subtyping check, we can ignore ϵ_1 and ϵ_2. In our formulation of subtyping, type constructors like arrows and records distribute over intersections, e.g. $A \to B \,\&\, C$ is equivalent to $(A \to B) \,\&\, (A \to C)$. Such distributivity first appeared in the system proposed by Barendregt et al. [1983] and is supported by λ_i^+ and F_i^+. We follow the subtyping algorithm design in λ_i^+ [Huang et al. 2021]. It distinguishes types that have a form equivalent to intersection types from others. Such types are called *splittable types* and can be separated into two via *type splitting*, which is defined in Fig. 16. For example, $A \to B \lhd A \to B \,\&\, C \rhd A \to C$ represents that $A \to B \,\&\, C$ is equivalent to the intersection of $A \to B$ and $A \to C$. The notation A° stands for types that are not splittable, which are called *ordinary types*.

In type splitting, the two split types are outputs. However, they are then used as inputs in the *coercive merging* judgment $\epsilon_1 : A \rhd C \lhd \epsilon_2 : B \rightsquigarrow \epsilon$, as defined in Fig. 16. If we omit ϵ_1 and ϵ_2 in this judgment, coercive merging characterizes the same relation as type splitting. In other words, removing $B_1 \lhd B \rhd B_2$ from rule S-Split does not change the idea of subtyping. We retain it to better represent the information flow in the subtyping algorithm: in rule S-Split, after being generated from type splitting, B_1 and B_2 are used to coerce the same term e individually, and the coerced results e_1 and e_2 are merged back, guided by the types. Note that, in this process, it is possible to duplicate terms and lead to duplicate labels in the corresponding target record terms.

Soundness of elaboration. The semantics of our source calculus is given via an elaboration, which reflects the compilation of CP. We establish our type-safety proofs on (1) the type safety of our target calculus, and (2) the soundness of elaboration, which connects the source calculus to the target calculus. Specifically, for every well-typed source expression, the typing derivation produces a target term, and we prove that the target term is well-typed in the record calculus. In addition, its type is equivalent to the translated type of the original source expression. The soundness of elaboration is based on the soundness of coercive subtyping, distributive application, and distributive projection, which guarantee that the terms generated from these judgments have the desired types. Note that the premises of these soundness lemmas are coarser than their conclusion: the actual type of the input term does not have to be equivalent to the annotated type. For example, in $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$, ϵ_1 only needs to have a subtype of $\|A\|$ for ϵ_2 to be correctly typed. This is because, in these coercive relations, the input terms are always used for projection (e.g. in rules S-Arrow and S-Rcd) but never for concatenation. As long as the input terms have sufficient fields, other fields that they have are unimportant.

Theorem 5.11 (Elaboration soundness). *We have that:*

- *If $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$ and $\Delta \vdash \epsilon_1 : \mathcal{A}$ and $\mathcal{A} \subseteq \|A\|$, then $\exists \mathcal{B}, \Delta \vdash \epsilon_2 : \mathcal{B}$ and $\mathcal{B} \approx \|B\|$.*
- *If $\epsilon_1 : A \bullet \epsilon_2 : B \rightsquigarrow \epsilon_3 : C$ and $\Delta \vdash \epsilon_1 : \mathcal{A}$ and $\mathcal{A} \subseteq \|A\|$ and $\Delta \vdash \epsilon_2 : \mathcal{B}$ and $\mathcal{B} \subseteq \|B\|$, then $\exists \mathcal{C}, \Delta \vdash \epsilon_3 : \mathcal{C}$ and $\mathcal{C} \approx \|C\|$.*
- *If $\epsilon_1 : A \bullet \{\ell\} \rightsquigarrow \epsilon_2 : B$ and $\Delta \vdash \epsilon_1 : \mathcal{A}$ and $\mathcal{A} \subseteq \|A\|$, then $\exists \mathcal{B}, \Delta \vdash \epsilon_2 : \mathcal{B}$ and $\mathcal{B} \approx \|B\|$.*
- *If $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow \epsilon$ then $\exists \mathcal{A}, |\Gamma| \vdash \epsilon : \mathcal{A}$ and $\mathcal{A} \approx \|A\|$.*

5.3 Duplicates in Translation and Coherence of Subtyping

With the disjointness constraint enforced in rule Ela-Merge, all elaborated records originating from that rule have distinct labels. However, we still have to take duplicates into account because they can be generated by coercive subtyping. For example, $\epsilon : \mathbb{Z} <: \mathbb{Z} \,\&\, \mathbb{Z} \rightsquigarrow \epsilon +\!\!+\, \epsilon$ duplicates ϵ. Note that given ϵ_1, A, and B, there could be more than a single possible ϵ_2, generated by different derivations of subtyping $\epsilon_1 : A <: B \rightsquigarrow \epsilon_2$. Therefore, it is possible to have $\epsilon_1 : A <: B \,\&\, B \rightsquigarrow \epsilon_2 \mathbin{+\!\!+} \epsilon_3$ where ϵ_2 and ϵ_3 are different. In the proof of Theorem 5.11, we show such results do not violate type well-formedness: ϵ_2 and ϵ_3 have equivalent types. Moreover, we conjecture that ϵ_2 and ϵ_3 have the same behavior, since similar results have been proven in the past (semantic coherence or determinism) for variants of λ_i^+ [Bi et al. 2018; Huang et al. 2021]. Because the disjointness restriction in rule Ela-Merge ensures that semantically different terms with equivalent types cannot be introduced into one merge, it is sufficient to distinguish terms by the type indices.

Past coherence results. The technical reason for our coercive subtyping not producing a unique result is that we allow types like $A_1 \,\&\, A_2$ even if a non-top-like type B exists such that $A_1 <: B$ and $A_2 <: B$ both hold, leading to two different subtyping derivation paths for $A_1 \,\&\, A_2 <: B$. One way to ensure the uniqueness of coercions, as utilized by previous work, is to reject such intersection types via a disjointness constraint in type well-formedness [Alpuim et al. 2017]. If intersection types are restricted in this way, we do not even need to worry about duplicates at all. However, unrestricted intersection types are more expressive and are required when encoding bounded polymorphism [Xie et al. 2020]. Moreover, imposing a disjoint constraint on all intersections significantly complicates the proof of type soundness [Alpuim et al. 2017]. That is why all subsequent work [Bi et al. 2018, 2019; Fan et al. 2022; Huang et al. 2021], besides ours, relaxed the restriction on intersections.

For our source calculus to be coherent, it is necessary to ensure that all the coercions generated from the same subtyping judgment are equivalent. Proving this property is challenging, especially considering the main focus we had when designing the formal calculi is to justify the usefulness of the compilation, for which the efficiency is more important. In other words, many design choices in the formalization are driven by efficiency considerations, rather than by considerations for making a proof of coherence easier. For instance, we use a novel and non-standard form of coercive subtyping, which sacrifices the ability to do induction on the coercion structure. Nevertheless, the coherence of the subtyping relation in previous work (and the determinism of the casting semantics implied by subtyping) has already been proven [Bi et al. 2018; Huang et al. 2021]. Although our setting differs slightly due to our use of a different target language, the previous results about coherence provide us confidence that the elaboration here is also coherent.

Determinism in a direct operational semantics. Huang et al.'s variant of λ_i^+ uses a direct operational semantics where annotations trigger subtyping checks and act as upcasts at run time, directly manipulating source values. The upcasting process mirrors the approach of algorithmic subtyping, which is similar to how coercions are generated in our subtyping judgments. For instance, the expression $1 : \mathbf{Int} \,\&\, \mathbf{Int}$ evaluates to $1 \,,\, 1$. When an integer is expected, either component can be selected. This type system permits duplicate components in merges. The operational semantics of this variant has been proven to be deterministic and type-safe, providing evidence that no ambiguity arises from subtyping when using disjoint merges.

Coherence for an elaboration semantics. Closer to our work, Bi et al.'s variant of λ_i^+, also known as NeColus, employs an elaboration semantics that is proven to be coherent. The NeColus calculus covers the same set of expressions as our source calculus and also features a syntax-directed bidirectional type system. It uses a different algorithm to decide subtyping, and provides a formulation in declarative style, which is equivalent to ours (see Huang et al.'s work for a formalization of this result). Most of the typing rules are also the same as ours, including the rule for merges and the subsumption rule. The rules for lambda abstraction, application, and record projection are slightly different in terms of requiring more or fewer type annotations, and NeColus does not have the separate relations for distributive application and projection.

The main difference between Bi et al.'s elaboration and ours, is that the target language for NeColus is a different calculus called λ_c. λ_c is a variant of the simply typed λ-calculus extended with records, products, and explicit coercions. In NeColus, merges are translated into pairs. For example, $1 \,,\, \textbf{true}$ is translated into $\langle 1, \textbf{true} \rangle$. In the coherence theorem, Bi et al. prove that such elaboration always leads to equivalent terms in the target language. Their proofs show that the duplication that arises in coercive subtyping does not cause ambiguity in the target language.

Before discussing the coherence proof, let us compare the two elaboration frameworks with some examples. Both λ_c and our λ_r only include the integer type $\mathbb{Z}$ as a representative of primitive types, but we will use **Bool** and **Int** in our examples for demonstration. Besides, we replace the coercions in λ_c by lambda terms and simplify all the elaboration results for easier comparison. Furthermore, we include JavaScript code to show the same situation in the code generated by our compiler.

Example 1. Consider the source expression:

$$1 : \textbf{Int} \,\&\, \textbf{Int} : \textbf{Int}$$

The translation to λ_r is unique:

$$\{|\textbf{Int}| \Rightarrow \{|\textbf{Int}| \Rightarrow 1;\ |\textbf{Int}| \Rightarrow 1\}.|\textbf{Int}|\}$$

The translation to λ_c has multiple possibilities, including:

$$(\lambda x.\ x.\mathsf{fst})\ \langle 1, 1 \rangle \qquad\qquad (\lambda x.\ x.\mathsf{snd})\ \langle 1, 1 \rangle$$

This example illustrates a challenging situation that involves two steps: first creating a term corresponding to type $\textbf{Int} \,\&\, \textbf{Int}$, and then selecting a component of type **Int**. In λ_r, the duplication in the first step causes a label conflict, and the second step is deterministic because of the overriding semantics; while in λ_c, it is the second step that brings potential ambiguity. With pairs serving as the target for merges in λ_c, every component in merges can be identified and extracted by position. The position information is analyzed from types during the elaboration. If the merge consists of two terms of the same type, the two positions can be used interchangeably. Therefore, there are two possible coercions that can upcast $\textbf{Int} \,\&\, \textbf{Int}$ to **Int**, namely $\lambda x.\ x.\mathsf{fst}$ and $\lambda x.\ x.\mathsf{snd}$. Note that, in a calculus with pairs, these two functions are clearly semantically different since they can be applied to a pair argument like $\langle 1, 2 \rangle$, which would produce two different results. But the point is that such pairs with different elements of the same type are never produced by the elaboration, so these two coercions behave identically for the *pairs that can be generated from the elaboration*.

This idea applies, more generally, to types with the same type index in our elaboration, and not just syntactically equal types. Types with the same type index are mutual subtypes and can interchange with each other in subtyping derivation. Another observation from this example is that, whenever there is an overriding in λ_r, there will be multiple possibilities in the translation to λ_c.

Finally, the compiled JavaScript code (without optimization) is as follows:

```
const $1 = {};  $1.int = 1;  $1.int = 1;
const $2 = {};  $2.int = $1.int;
```

The JavaScript code generated by our compiler shares the same overriding semantics as λ_r: `$1.int` is assigned twice, and the second assignment overrides the first one. Note that we have implemented several optimizations in our compiler, including eliminating redundant coercions, so the actual code generated by our compiler is more concise than the one shown here. Since $\textbf{Int} \,\&\, \textbf{Int}$ and **Int** are equivalent types, no coercions will be inserted in the optimized code. We keep the code unoptimized here to better illustrate the correspondence between the JavaScript code and the λ_r term.

Example 2. Consider another source expression:

$$(\lambda f.\, f) : (\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}) \to \mathbf{Int} \to \mathbf{Int}$$

This time the translation to λ_r has two possibilities. Here we use A to denote the type $(\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}) \to \mathbf{Int} \to \mathbf{Int}$:

$$\{|A| \Mapsto \lambda f.\, \{|\mathbf{Int} \to \mathbf{Int}| \Mapsto \lambda x.\, \{|\mathbf{Int}| \Mapsto ((f.|\mathbf{Int} \to \mathbf{Int}|)\, \{|\mathbf{Int}| \Mapsto x.|\mathbf{Int}|\}).|\mathbf{Int}|\}\}\}$$

$$\{|A| \Mapsto \lambda f.\, \{|\mathbf{Int} \to \mathbf{Int}| \Mapsto \lambda x.\, \{|\mathbf{Int}| \Mapsto ((f.|\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}|)\, \{|\mathbf{Int}| \Mapsto x.|\mathbf{Int}|\}).|\mathbf{Int}|\}\}\}$$

The two possibilities correspond to the translation to λ_c as follows:[6]

$$\lambda f.\, \lambda x.\, f.\mathsf{fst}\; x \qquad\qquad \lambda f.\, \lambda x.\, (f.\mathsf{snd}\; x).\mathsf{fst}$$

This is a case where λ_r has multiple syntactically different translation results. The higher-order function expects a parameter f, which is a record in λ_r that has two fields with label $|\mathbf{Int} \to \mathbf{Int}|$ and label $|\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}|$. These two type indices are different, but the two types are not disjoint. Therefore, if their common supertype is desired in a source context, both fields can be selected. For the purpose of coherence, the two results originating from both sides should behave the same; that is to say, they should have equivalent semantics for the overlapping part of their types (i.e. $\mathbf{Int} \to \mathbf{Int}$). This is also needed for λ_c: the two translated terms should only apply to a pair of functions that have the same behavior, if we only consider the integer part in their return results. Because of disjointness, the only way to create a term with type $(\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool})$ is using one lambda abstraction, such as $(\lambda x.\, x \,,, \,\mathbf{true}) : \mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool} : (\mathbf{Int} \to \mathbf{Int}) \,\&\, (\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool})$. It does not type-check to use a merge of two functions with types $\mathbf{Int} \to \mathbf{Int}$ and $\mathbf{Int} \to \mathbf{Int} \,\&\, \mathbf{Bool}$ as these two types are not disjoint. Therefore, the function that can be selected by the multiple coercions is the same function, which was just duplicated twice.

The compilation to JavaScript may have two possible versions as well:

```
const $1 = {};
$1.fun_fun_int = function ($f) {
  const $2 = {};
  $2.fun_int = function ($3) {
    const $4 = {};
    $4.int = $3.int;
    const $5 = $f.fun_int($4);
    const $6 = {};
    $6.int = $5.int;
    return $6;
  };
  return $2;
};
```

```
const $1 = {};
$1.fun_fun_int = function ($f) {
  const $2 = {};
  $2.fun_int = function ($3) {
    const $4 = {};
    $4.int = $3.int;
    const $5 = $f['fun_(bool&int)']($4);
    const $6 = {};
    $6.int = $5.int;
    return $6;
  };
  return $2;
};
```

Note that the only difference is the value of `$5`: one is created by applying `$f.fun_int` and the other by applying `$f['fun_(bool&int)']` (n.b. `$f.fun_(bool&int)` does not work because the label

[6]The λ_c terms look much simpler because they are derived from declarative subtyping for brevity. If we derive the coercions from algorithmic subtyping, the terms will be more complicated:

$$\lambda x_1.\, (\lambda x_2.\, (\lambda x_3.\, \lambda x_4.\, (x_3\; x_4))\; ((\lambda x_5.\, x_5.\mathsf{fst})\; x_2))\; x_1$$

$$\lambda x_1.\, (\lambda x_2.\, (\lambda x_4.\, \lambda x_5.\, ((\lambda x_6.\, x_6.\mathsf{fst})\; (x_4\; x_5)))\; ((\lambda x_7.\, x_7.\mathsf{snd})\; x_2))\; x_1$$

contains special characters). The two versions will behave identically, so it does not matter which one is actually generated. Similarly to the situation in λ_r, as long as the CP code is well-typed and is compiled to JavaScript by our compiler, `$f.fun_int` and `$f['fun_(bool&int)']` originate from the same function and behave the same as only the integer part of the return value (i.e. `$5.int`) is used. Again, the JavaScript code is deliberately kept unoptimized to show the correspondence with the λ_r terms. In the optimized code for the first version, for instance, the creation of `$4` and `$6` can be eliminated.

5.4 Coherence Proof for NeColus and Its Adaptation to Our Source Language

The coherence result in NeColus is established using a semantic approach based on *logical relations* [Biernacki and Polesiuk 2015; Tait 1967]. Here we provide a summary of the key steps in that coherence proof. We will use E, V, τ, and Δ to represent the expressions, values, types, and typing contexts in λ_c, the target language for NeColus.

Coherence via contextual equivalence. The coherence theorem proven for NeColus is as follows:

Theorem 5.12 (Coherence of NeColus). *If* $\Gamma \vdash e \Leftrightarrow A$ *then* $\Gamma \vdash e \simeq_{ctx} e : A$.

Here e is a source expression, so it is to say that a well-typed source expression is contextually equivalent to itself. The notation $\Leftrightarrow$ stands for both bidirectional typing modes, namely inference ($\Rightarrow$) and checking ($\Leftarrow$). Contextual equivalence is defined as:

Definition 5.13 (Contextual equivalence in NeColus). $\Gamma \vdash e_1 \simeq_{ctx} e_2 : A \triangleq$

$$\forall E_1\, E_2\, C\, D,\ \textit{if}\ \Gamma \vdash e_1 \Leftrightarrow A \rightsquigarrow E_1\ \textit{and}\ \Gamma \vdash e_2 \Leftrightarrow A \rightsquigarrow E_2\ \textit{and}\ C : (\Gamma \Leftrightarrow A) \mapsto (\cdot \Leftrightarrow \mathbb{Z}) \rightsquigarrow D$$
$$\textit{then}\ D[E_1] \simeq D[E_2].$$

The intuition behind the definition is that two elaborated terms (E_1 and E_2) should be considered equivalent if, for any well-typed contexts (that could be generated during the elaboration), plugging either of them in makes no difference to the final evaluation result. Here C and D stand for source and target *expression contexts*, respectively. An expression context is an expression that contains a hole $[\cdot]$. The typing judgment for contexts $C : (\Gamma \Leftrightarrow A) \mapsto (\cdot \Leftrightarrow \mathbb{Z}) \rightsquigarrow D$ means that, given any well-typed NeColus expression $\Gamma \vdash e \Leftrightarrow A$, we have $\cdot \vdash C[e] \Leftrightarrow \mathbb{Z}$, and the source context C corresponds to a target context D in elaboration, which, similarly, make $D[E_1]$ and $D[E_2]$ have type $\mathbb{Z}$. In this definition, $\simeq$ stands for Kleene equality. That is to say, $D[E_1]$ and $D[E_2]$ evaluate to the same integer. Here the definition intentionally excludes the contexts that cannot be obtained from a well-typed NeColus expression. For example, the source expression $(\lambda x.\, x) : \mathbb{Z} \,\&\, \mathbb{Z} \to \mathbb{Z}$ can be elaborated into $\lambda x.\, x.\mathrm{fst}$ or $\lambda x.\, x.\mathrm{snd}$. To judge whether they have the same contribution to computation, we should not consider the target expression context $[\cdot]\langle 1, 2\rangle$, that is, applying the elaborated function to $\langle 1, 2\rangle$, because its corresponding source term violates the disjointness restriction and thus is not well-typed.

Heterogeneous logical relations. NeColus introduces two heterogeneous logical relations that relates values and terms, respectively, in the target language λ_c, as shown in Fig. 17. The logical relation is designed to capture the intuition of coherent expressions – those that are safe to coexist in pairs, as they are unambiguous in every valid context.

First, $\mathcal{V}[\![\tau_1; \tau_2]\!]$ relates all duplicate values. For example, $(1, 1) \in \mathcal{V}[\![\mathbb{Z}; \mathbb{Z}]\!]$ because it never leads to ambiguity. Second, it is proven that all disjoint values are also related. For example, as long as we have $\mathbb{Z} * \{\ell : \mathbb{Z}\}$, we also have $(1, \{\ell = 1\}) \in \mathcal{V}[\![\mathbb{Z}; \{\ell : \mathbb{Z}\}]\!]$. For product types, the relation distributes over the product constructor $\times$. This reflects the disjointness of intersection types, that is, $A \,\&\, B * C$ if and only if $A * C$ and $B * C$. Finally, $\mathcal{E}[\![\tau_1; \tau_2]\!]$ states that E_1 and E_2 are related if they

$$
\begin{aligned}
(V_1, V_2) \in \mathcal{V}[\![\mathbb{Z};\mathbb{Z}]\!] &\triangleq \exists n,\ V_1 = V_2 = n \\
(V_1, V_2) \in \mathcal{V}[\![\tau_1 \to \tau_2; \tau_1' \to \tau_2']\!] &\triangleq \forall (V, V') \in \mathcal{V}[\![\tau_1; \tau_1']\!],\ (V_1\, V, V_2\, V') \in \mathcal{E}[\![\tau_2; \tau_2']\!] \\
(\{\ell = V_1\}, \{\ell = V_2\}) \in \mathcal{V}[\![\{\ell : \tau_1\}; \{\ell : \tau_2\}]\!] &\triangleq (V_1, V_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] \\
(\langle V_1, V_2\rangle, V_3) \in \mathcal{V}[\![\tau_1 \times \tau_2; \tau_3]\!] &\triangleq (V_1, V_3) \in \mathcal{V}[\![\tau_1; \tau_3]\!] \wedge (V_2, V_3) \in \mathcal{V}[\![\tau_2; \tau_3]\!] \\
(V_3, \langle V_1, V_2\rangle) \in \mathcal{V}[\![\tau_3; \tau_1 \times \tau_2]\!] &\triangleq (V_3, V_1) \in \mathcal{V}[\![\tau_3; \tau_1]\!] \wedge (V_3, V_2) \in \mathcal{V}[\![\tau_3; \tau_2]\!] \\
(V_1, V_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!] &\triangleq \textbf{true} \quad \text{otherwise} \\
(E_1, E_2) \in \mathcal{E}[\![\tau_1; \tau_2]\!] &\triangleq \exists V_1\, V_2,\ E_1 \longrightarrow^* V_1 \wedge E_2 \longrightarrow^* V_2 \wedge (V_1, V_2) \in \mathcal{V}[\![\tau_1; \tau_2]\!]
\end{aligned}
$$

Fig. 17. Logical relations for λ_c (the target calculus of NeColus).

evaluate to related values. The relation is then lifted to open terms via a semantic interpretation of typing contexts, and the logical equivalence is defined in a standard way: two open terms are related if every pair of related closing substitutions make them related. $\Delta \vdash E_1 \simeq_{log} E_2 : \tau$ denotes that two well-typed expressions are logically equivalent with respect to the same typing context and the same type.

Fundamental property. In NeColus, a fundamental property is proven, stating that any two λ_c terms elaborated from the same NeColus expression are related by the logical relation. Here $\|\Gamma\|$ and $\|A\|$ stand for the translation of typing contexts and types from NeColus to λ_c.

Theorem 5.14 (Fundamental property of NeColus). *If* $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow E_1$ *and* $\Gamma \vdash e \Leftrightarrow A \rightsquigarrow E_2$, *then* $\|\Gamma\| \vdash E_1 \simeq_{log} E_2 : \|A\|$.

Note that $\Delta \vdash E \simeq_{log} E : \tau$ does not hold for every well-typed target E. For example, the logical relation does not consider $\langle 1, 2\rangle$. Since there is no possible elaboration (the source expression $1 ,, 2$ violates the disjoint constraint), this does not prevent the fundamental property. Actually the limitation on coherent products helps the logical relation to accept some semantically equivalent terms like $\lambda x.\, x.\text{fst}$ and $\lambda x.\, x.\text{snd}$: they are two translations of $(\lambda x.\, x) : \textbf{Int} \,\&\, \textbf{Int} \to \textbf{Int}$. With the assumption that the two integers in the pair argument are related by the logical relation, choosing either one leads to the same result.

Soundness. Since the fundamental property has shown that different elaborations of the same NeColus expression are logically equivalent, the remaining step is to show that logical equivalence is sound with respect to contextual equivalence.

A compatibility lemma for coercions is proven during the establishment of the fundamental property: if two terms are related by the logical relation, after applying a coercion to one term, they are still related. Based on this lemma, we can prove that the logical relation is preserved by all well-typed NeColus contexts, including the contexts that coverts a term to an integer result, for which the logical relation implies Kleene equality. Then it is straightforward to show that the logical relation is sound.

Theorem 5.15 (Soundness w.r.t. contextual equivalence in NeColus). *If* $\Gamma \vdash e_1 \Leftrightarrow A \rightsquigarrow E_1$ *and* $\Gamma \vdash e_2 \Leftrightarrow A \rightsquigarrow E_2$ *and* $\|\Gamma\| \vdash E_1 \simeq_{log} E_2 : \|A\|$, *then* $\Gamma \vdash e_1 \simeq_{ctx} e_2 : A$.

Finally, the coherence theorem follows directly from the fundamental property and the soundness result. In other words, we can conclude that any two λ_c terms elaborated from the same NeColus expression are contextually equivalent.

$$
\begin{aligned}
(v_1, v_2) \in \mathcal{V}_\nabla[\![\mathbb{Z};\mathbb{Z}]\!] &\triangleq \exists n,\ v_1 = v_2 = n \\
(v_1, v_2) \in \mathcal{V}_\nabla[\![\mathcal{A}_1 \to \mathcal{B}_1;\mathcal{A}_2 \to \mathcal{B}_2]\!] &\triangleq \forall (v, v') \in \mathcal{V}_\nabla[\![\mathcal{A}_1;\mathcal{A}_2]\!], \\
&\ (v_1\ v, v_2\ v') \in \mathcal{E}_\nabla[\![\mathcal{B}_1;\mathcal{B}_2]\!] \\
(\{\ell_1 \Mapsto v_1\}, \{\ell_2 \Mapsto v_2\}) \in \mathcal{V}_\nabla[\![\{\ell_1 \Mapsto \mathcal{A}_1\};\{\ell_2 \Mapsto \mathcal{A}_2\}]\!] &\triangleq (v_1, v_2) \in \mathcal{V}_\nabla[\![\mathcal{A}_1;\mathcal{A}_2]\!] \quad \text{if } \ell_1 = \ell_2 = \overline{\mathbb{Z}} \\
&\ \text{or } (\exists Ts_i,\ \ell_1 = \overline{Ts_1 \to Ts_2} \land \ell_2 = \overline{Ts_3 \to Ts_4}) \\
&\ \text{or } (\exists \ell\ Ts_i,\ \ell_1 = \overline{\{\ell : Ts_1\}} \land \ell_2 = \overline{\{\ell : Ts_2\}}) \\
(\{\ell_1 \Mapsto v_1;\ \dots;\ \ell_n \Mapsto v_n\}, v') \in \mathcal{V}_\nabla[\![\{\ell_1 \Mapsto \mathcal{A} \mid \rho\};\mathcal{B}]\!] &\triangleq (\{\ell_1 \Mapsto v_1\}, v') \in \mathcal{V}_\nabla[\![\{\ell_1 \Mapsto \mathcal{A}\};\mathcal{B}]\!] \land \\
&\ (\{\ell_2 \Mapsto v_2;\ \dots;\ \ell_n \Mapsto v_n\}, v') \in \mathcal{V}_\nabla[\![\rho;\mathcal{B}]\!] \\
(v', \{\ell_1 \Mapsto v_1;\ \dots;\ \ell_n \Mapsto v_n\}) \in \mathcal{V}_\nabla[\![\mathcal{B};\{\ell_1 \Mapsto \mathcal{A} \mid \rho\}]\!] &\triangleq (v', \{\ell_1 \Mapsto v_1\}) \in \mathcal{V}_\nabla[\![\mathcal{B};\{\ell_1 \Mapsto \mathcal{A}\}]\!] \land \\
&\ (v', \{\ell_2 \Mapsto v_2;\ \dots;\ \ell_n \Mapsto v_n\}) \in \mathcal{V}_\nabla[\![\mathcal{B};\rho]\!] \\
(v_1, v_2) \in \mathcal{V}_\nabla[\![\mathcal{A}_1;\mathcal{A}_2]\!] &\triangleq \textbf{true} \quad \text{otherwise} \\
(\epsilon_1, \epsilon_2) \in \mathcal{E}_\nabla[\![\mathcal{A}_1;\mathcal{A}_2]\!] &\triangleq \exists v_1\ v_2,\ \epsilon_1 \longrightarrow^* v_1 \ \land\ \epsilon_2 \longrightarrow^* v_2 \land \\
&\ \quad (v_1, v_2) \in \mathcal{V}_\nabla[\![\mathcal{A}_1;\mathcal{A}_2]\!]
\end{aligned}
$$

Fig. 18. Logical relations for λ_r (our target calculus).

Adapting the proof to our source language: a sketch. While our compilation scheme makes some different design choices for efficiency, it essentially shares the same principle of coherence with NeColus. Here we provide a sketch of how to adapt the ideas from the NeColus proof to our work, although we do not provide a full proof.

We can define two logical relations for values ($\mathcal{V}_\nabla[\![\mathcal{A};\mathcal{B}]\!]$) and terms ($\mathcal{E}_\nabla[\![\mathcal{A};\mathcal{B}]\!]$) in λ_r, as presented in Fig. 18. In this definition, we relate two records if they do not cause ambiguity under any contexts that can be elaborated from our source calculus. That is to say, for two records to be related, any pair of fields from them, as long as their labels correspond to overlapping source types, must have related fields. For example, relating $\{|\textbf{Int} \to \textbf{Int} \,\&\, \textbf{Bool}| \Mapsto v_1\}$ and $\{|\textbf{Int} \to \textbf{Int}| \Mapsto v_2\}$ in the logical relation requires v_1 and v_2 to be related. Considering a source expression context $[\cdot] : \textbf{Int} \to \textbf{Int}$, it applies to both records and leads to two projections that extract v_1 and v_2, respectively. So v_1 and v_2 should be equivalent. Besides, disjoint terms are also related. For example, $\{|\{\ell_1 : A\}| \Mapsto v_1\}$ and $\{|\{\ell_2 : B\}| \Mapsto v_2\}$ correspond to two source records of type $\{\ell_1 : A\}$ and type $\{\ell_2 : B\}$. If $\ell_1 = \ell_2$, their relation should be decided by the fields; if $\ell_1 \neq \ell_2$, they are always related because of disjointness. The relation of terms can be lifted to open terms like in NeColus. We expect the logical equivalence derived from the logical relation to be compatible with our coercive subtyping, and a fundamental property should follow. The main rationale is that the overlapping part must have the same origin, restricted by type disjointness. What coercions do is to decompose and recompose the behaviors according to the types (labels). Disjoint terms will be distinguished clearly in the process. A field of $|\{\ell_1 : A\}|$, for example, will not be used to build a field of $|\{\ell_2 : A\}|$ if $\ell_1 \neq \ell_2$. No λ_i^+ context should violate the derived logical equivalence. We anticipate a similar theorem asserting that logical equivalence implies contextual equivalence, provided that contextual equivalence is defined in a manner analogous to NeColus. Consequently, the coherence theorem would follow, in the style of Theorem 5.12.

```
const $1 = { int: 48 };
const $2 = { bool: true };
const $3 = { ...$1, ...$2 };
const $4 = { int: 32 };
const $5 = $chr.fun_char($4);
const $6 = { ...$3, ...$5 };
```

(a) Before optimization: 6 objects.

```
const $1 = {};
$1.int = 48;
$1.bool = true;
const $2 = {};
$2.int = 32;
$chr.fun_char($2, $1);
```

(b) After optimization: 2 objects.

Fig. 19. Simplified JavaScript code for $48 ,, \textbf{true} ,, \texttt{chr}\ 32$.

6 Implementation Details

This section introduces our concrete implementation of the CP compiler that targets JavaScript. The full set of compilation rules can be found in Appendix A. Our implementation is available in the supplementary materials.

6.1 From Elaboration Semantics to JavaScript Code Generation

In the elaboration semantics presented in Section 5, we use a λ-calculus with extensible records as the target. In the actual compilation, records are modeled as JavaScript objects, and type indices are realized as objects' property names. For example, $48 ,, \textbf{true}$ compiles to `{ int: 48, bool: true }` in JavaScript. Besides, record concatenation can be directly represented by object merging using the spread syntax like `{ ...obj1, ...obj2 }`.

As we have mentioned, the formalized target language is a functional calculus, but JavaScript is an imperative language. The mismatch in programming paradigms is an important consideration when implementing the compilation to JavaScript. We consider two designs of target forms: one is based on *static single assignment* (SSA) [Cytron et al. 1991], and the other is based on *destination-passing style* (DPS) [Shaikhha et al. 2017]. We eventually choose the latter due to performance reasons, which we will explain next.

Reducing intermediate objects. In our initial design based on the SSA form, every subterm in a merge creates a new object in the compiled JavaScript code. Consequently, there will be too many intermediate objects that are useless. For example, consider the merge $48 ,, \textbf{true} ,, \texttt{chr}\ 32$, where `chr` is a function that converts an integer to a character, and the compiled function has been stored in `$chr`. We need to create six JavaScript objects in the SSA-based form, as shown in Fig. 19a. As mitigation, we adopt a new design based on DPS. We just create one object for the merge (e.g. `$1` in Fig. 19b) and pass the variable down to subterms to update their corresponding properties. To further prevent the function application (e.g. `chr` 32) from creating any intermediate object, we add an extra parameter when compiling all functions, including `chr`. The destination object (e.g. `$1`) is passed to the compiled function as the last argument, and the function body will directly write to that argument instead of creating a new object. In other words, `$1` is an output parameter while `$2` is an input. As a result, we reduce the creation of six objects to only two objects and avoid two object concatenations, as shown in Fig. 19b.

6.2 Parametric Polymorphism

The compilation of polymorphic terms is not formalized in Section 5, so here we introduce our solution in more detail. The challenge posed by parametric polymorphism is mainly because type arguments are unknown until type application. For those terms whose types contain free variables,

```
const $poly = {};
$poly['forall_fun_(1&int)'] = function ($A, $1) {
  $1['fun_' + toIndex([ ...$A, 'int' ])] = function ($x, $2) {
    for (const $A$elem of $A) $2[$A$elem] = $x[$A$elem];
    $2.int = 48;
  };
};
```

(a) `poly = /\(A * Int). \(x: A) → x , 48.`

```
const $3 = {};  const $4 = {};
$poly['forall_fun_(1&int)']([ 'string', 'bool' ], $4);
const $5 = {};  $5.string = 'foo';  $5.bool = true;
$4['fun_(bool&int&string)']($5, $3);
```

(b) `poly @(String & Bool) ("foo" , true).`

```
function toIndex(tt) {
  const ts = tt.sort().filter((t, i) => t === 0 || t !== tt[i-1]);
  if (ts.length === 1) return ts[0];
  else return '(' + ts.join('&') + ')';
};
```

(c) `toIndex`: an auxiliary function for generating type indices at run time.

Fig. 20. Simplified JavaScript code for polymorphic terms.

we can only generate type indices at run time. For example, Fig. 20a shows the simplified JavaScript code for the following definition in CP:

```
poly = /\(A * Int). \(x: A) → x , 48;
```

Here the notation `/\(A * Int)` represents a type parameter `A` bound by a Λ-function where `A` is disjoint with `Int`. The `poly` function can be typed as $\forall A * \mathbf{Int}.\, A \rightarrow A \,\&\, \mathbf{Int}$. We employ a *de Bruijn index* [de Bruijn 1972] to represent the bound variable `A`, so the Λ-function's type index is `"forall_fun_(1&int)"`. In contrast, the inner λ-function's type index is more intriguing. Before we introduce our approach, let us first see an example of applying `poly`:

```
poly @(String & Bool) ("foo" , true)
```

The type parameter is instantiated with an intersection type (`String & Bool`), and Fig. 20b shows the simplified JavaScript code for the application. After `poly` is instantiated, the inner λ-function will be used with concrete type indices (e.g. `"fun_(bool&int&string)"`) instead of something like `"fun_(A&int)"`. To achieve this, we pass the instantiated type as an argument to the outer Λ-function. The argument is in the form of a JavaScript array, which consists of the component types in the case of an intersection type. The array is empty for top-like types, or it is a singleton array for non-intersection, non-top-like cases. We also predefine a `toIndex` function to help generate the type index based on the runtime instantiation, whose definition is shown in Fig. 20c. The `toIndex` function accepts an array of types and generates their intersection's type index. To fit in with the notion of equivalent types in Section 4.3, it will sort and deduplicate the component types. Now that some type indices are dynamically generated, we have to use `obj[toIndex($A)]` instead

```
const $1 = {};
$1['rcd_x:int'] =
    $1['rcd_y:int'];
```

(a) By value.

```
const $1 = {};
$1.__defineGetter__(
  'rcd_x:int',
  function () {
    return $1['rcd_y:int'];
  });
```

(b) By thunk.

```
const $1 = {};
$1.__defineGetter__(
  'rcd_x:int',
  function () {
    delete this['rcd_x:int'];
    return this['rcd_x:int'] =
     $1['rcd_y:int'];
  });
```

(c) By memoized thunk.

Fig. 21. Simplified JavaScript code for `{ x = this.y }`.

of `obj.name` directly. Such a feature is called first-class labels [Leijen 2004] and is supported in JavaScript via computed property names with brackets. The situation in Fig. 20a is a bit more complicated because the type of the inner λ-function is $A \to A \,\&\, \mathbf{Int}$, so the dynamically computed type index is `"fun_" + toIndex([ ...$A, "int" ])`.

As illustrated above, the compiled code for polymorphic definitions incurs overhead due to the dynamic computation of type indices. In mainstream languages, parametric polymorphism is implemented via either *erasure* [Igarashi et al. 2001] or *monomorphization* [Griesemer et al. 2020]. Our compilation scheme cannot erase type information at run time, so monomorphization is a potential direction for improving the performance of polymorphic code. We leave this for future work.

6.3 Lazy Evaluation

In the most recent work by Fan et al. [2022], F_i^+ is formalized as a call-by-name calculus to correctly model trait instantiation. A simple example is as follows:

```
type Rcd = { x: Int; y: Int };
new (trait [this: Rcd] ⇒ { x = this.y; y = 48 })
```

The program will not terminate if evaluated using the call-by-value strategy. That is why Fan et al. go for a call-by-name semantics and evaluate record fields lazily. However, a naive call-by-name implementation may evaluate the same record field more than once and cause a significant slowdown. Even with proper memoization (call-by-need), the performance of generated code is still not ideal as JavaScript does not support lazy evaluation natively. In our implementation, we employ a hybrid strategy: only self-annotated trait fields are lazily evaluated, and other language constructs including function applications are strictly evaluated. This approximates the semantics in conventional OOP languages, which are call-by-value in terms of initializing fields and calling methods, except for lazy fields.

To better illustrate different evaluation strategies for record fields, we show three code snippets generated for the field `{ x = this.y }`. Fig. 21a employs strict evaluation, but instead of non-termination, the issue here is that the field `y` is not yet available, so field `x` is unexpectedly assigned **`undefined`**. This issue severely limits self-references, and the code in Fig. 21a would be broken. Thus we need a better approach. Fig. 21b resolves the issue by adding a *thunk*. The field is wrapped in a getter, and thus the computation of `this.y` is delayed until the whole record is constructed with the field `y` available. But note that the getter is called every time the field is accessed. This is undesirable as the computation in the thunk would be triggered on every access to the field. We optimize this

$\boxed{\epsilon_1 : A <:^{\pm} B \rightsquigarrow \epsilon_2}$ *(Optimized coercive subtyping)*

$$\frac{A \doteqdot B}{\epsilon : A <:^{+} B \rightsquigarrow \epsilon}\ \textsc{S-Equiv} \qquad \frac{\epsilon : A <:^{-} C \rightsquigarrow \epsilon'}{\epsilon : A \,\&\, B <:^{\pm} C \rightsquigarrow \epsilon'}\ \textsc{S-AndL}$$

$$\frac{x : B_1 <:^{+} A_1 \rightsquigarrow \epsilon_1 \qquad (\epsilon.|A_1 \to A_2|)\, \epsilon_1 : A_2 <:^{+} B_2 \rightsquigarrow \epsilon_2}{\epsilon : A_1 \to A_2 <:^{\pm} B_1 \to B_2 \rightsquigarrow \{|B_1 \to B_2| \Mapsto \lambda x.\, \epsilon_2\}}\ \textsc{S-Arrow}$$

Fig. 22. Selected rules for optimized coercive subtyping.

by memoizing fields using *smart getters*.[7] As shown in Fig. 21c, once the field is evaluated, the getter is deleted, and the value is stored instead. Our implementation automatically detects self-annotated trait fields and applies the last approach to them; for other cases, the first approach is applied. By this means, self-references and trait instantiation in CP are correctly supported while maintaining good performance for other language constructs.

6.4 Important Optimizations

Eliminating redundant coercions. In the rules of coercive subtyping shown in Fig. 15, even $A <: A$ may go through a lot of rules when A is a complex intersection type. However, as long as we encounter subtyping between *equivalent* types, we do not need any coercion code. To implement this optimization, an immediate idea would be:

- Adding a special case to rule Ela-Sub such that if $A \doteqdot B$ then $\epsilon_1 = \epsilon_2$.

Unfortunately, this rule does not deal with many important cases. For instance, when checking $A \to B <: A \to C$, we would like to avoid applying coercions to the inputs of the function (since they have the same type). Therefore, besides the rule above, we may consider:

- Adding an extra rule (say S-Equiv in Fig. 22) such that if $A \doteqdot B$ then $\epsilon : A <: B \rightsquigarrow \epsilon$.

However, this idea is incorrect when an intersection type occurs on the left-hand side. For example, when upcasting 48 , **true** to type **Int**, the coercion is missing in the subtyping derivation:

$$\dfrac{\dfrac{\mathbf{Int} \doteqdot \mathbf{Int}}{\{\texttt{int} \Mapsto 48;\ \texttt{bool} \Mapsto \mathbf{true}\} : \mathbf{Int} <: \mathbf{Int} \rightsquigarrow \{\texttt{int} \Mapsto 48;\ \texttt{bool} \Mapsto \mathbf{true}\}}\ \textsc{S-Equiv}}{\{\texttt{int} \Mapsto 48;\ \texttt{bool} \Mapsto \mathbf{true}\} : \mathbf{Int} \,\&\, \mathbf{Bool} <: \mathbf{Int} \rightsquigarrow \{\texttt{int} \Mapsto 48;\ \texttt{bool} \Mapsto \mathbf{true}\}}\ \textsc{S-AndL}$$

Observing that rules S-AndL and S-AndR are the root cause of missing coercions, we add an extra flag that indicates the applicability of rule S-Equiv to fix the latter idea. As shown in Fig. 22, by default ($<:^{+}$) the optimization S-Equiv can apply, but it will be disabled ($<:^{-}$) in the derivations of rule S-AndL (as well as rule S-AndR), and re-enabled in derivations of rule S-Arrow (as well as rules S-All and S-Rcd). In some rules, the flag does not matter, so we use $<:^{\pm}$ to mean that both cases apply. The complete rules targeting JavaScript can be found in Appendix A.

For a simple example of the optimized code, we consider a CP function that takes a parameter of type **Int&Bool**, and we pass a merge of type **Bool&Int** as its argument:

```
(\(x:Int&Bool) → x:Int) (true , 48)
```

The compiled JavaScript for this function application is:

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get#smart_self-overwriting_lazy_getters

```
const $1 = {};
const $2 = {};  $2.fun_int = function ($x, $3) { $3.int = $x.int };
const $4 = {};  $4.bool = true;  $4.int = 48;
$2.fun_int($4, $1);
```

Although it goes through the rule A-Arrow to check the argument of type `Bool&Int` against its supertype `Int&Bool`, there is no coercion inserted for `$4`. This is just due to the elimination of coercions for subtyping between equivalent types ($\mathbf{Bool} \,\&\, \mathbf{Int} \fallingdotseq \mathbf{Int} \,\&\, \mathbf{Bool}$).

Optimizing record projection. In the formalization of F_i^+ by Fan et al. [2022], a record with more than one label can only be projected after a coercion is inserted to remove irrelevant labels. For example, `{x = 1; y = 2}.x` in CP has to be elaborated into `({x = 1},{y = 2} : {x: Int}).x` in F_i^+ to make the record projection work. If a similar technique is used in the CP compiler, there will be too many coercions that lead to poor performance. This flaw is because the formalization of F_i^+ reuses the rules of distributive application for record projection. However, the semantics for distributive projection is slightly different: we do not require that every component of a record can be projected by the same label. Therefore, we add rule JP-RcdNeq to safely ignore irrelevant fields. The complete rules have been formalized in Fig. 14. By implementing the new rules for distributive projection, the compiled JavaScript can directly handle projection for multi-field records.

Reducing object copying. Although the DPS-based design reduces the number of intermediate objects, it sometimes introduces unnecessary object copying. For example, consider a function `id` that takes a parameter of type `Double&Int&String` and returns it as is:

```
id (x: Double&Int&String) = x;
```

The compiled JavaScript code based on DPS is shown in Fig. 23a. The function always copies the fields from the parameter `$x` to the destination object `$1`. Such object copying is necessary if the result of the function call is part of a merge (e.g. id y , **true**), but it is inefficient otherwise (e.g. id $y + 1$). To optimize this, we do not pass a destination object to the function if the function call does not occur in a merge, as shown in Fig. 24a. The function call in a merge still follows the DPS design we discussed earlier, as shown in Fig. 24b. To distinguish these two cases, we add a dynamic check in the compiled function to avoid unnecessary copying when the destination (`$1`) is **`undefined`**, as shown in Fig. 23b.

Optional destination objects. We have avoided unnecessary copying when a destination is absent, but what about the dual case: how to avoid a fresh object when a destination is present? Since the body of the previous function `id` is just a variable, no fresh object is created in any case. Let us consider another function `con`, which returns a merge of **`false`** and `0` regardless of the input:

```
con (_: Top) = false,0;
```

The compiled JavaScript code is shown in Fig. 25. `$1 || {}` on the first line of the function body checks if the destination (`$1`) is present. If it is, `$2` is just an alias of `$1`; otherwise, a fresh object `{}` is created and assigned to `$2`. By this means, we avoid creating a fresh object if the destination is provided.

Avoiding boxing/unboxing. Although extensible records, or more specifically, JavaScript objects, serve as an excellent target for merges in CP, they are not so efficient when only primitive values and their computation are involved. For example, when compiling $1 + 2$ to JavaScript, the naive approach is shown in Fig. 26a. It would first create two objects for 1 and 2, then access their `"int"` fields to get the values, perform the addition, and finally create a new object for the result. These tedious wrapping/unwrapping of objects are similar to boxing/unboxing in Java and are

```
const $id = {};
$id['fun_(double&int&string)'] = function ($x, $1) {
  $1.double = $x.double;  $1.int = $x.int;  $1.string = $x.string;
};
```

(a) Before optimization: always copying fields from $x to $1.

```
const $id = {};
$id['fun_(double&int&string)'] = function ($x, $1) {
  if ($1) { $1.double = $x.double;  $1.int = $x.int;  $1.string = $x.string; }
  return $x;
};
```

(b) After optimization: no copying when $1 is **undefined**.

Fig. 23. Simplified JavaScript code for `id (x: Double&Int&String) = x`.

```
const $1 = {};
const $2 = $id['fun_(double&int&string)']($y);
const $3 = {};  $3.int = 1;
$1.int = $2.int + $3.int;
```

(a) id $y + 1$.

```
const $1 = {};
$id['fun_(double&int&string)']($y, $1);
$1.bool = true;
```

(b) id y , **true**.

Fig. 24. Simplified JavaScript code for applying id.

```
const $con = {};
$con['fun_(bool&int)'] = function ($_, $1) {
  $2 = $1 || {};
  $2.bool = false;  $2.int = 0;
  return $2;
};
```

Fig. 25. Simplified JavaScript code for `con (_: Top) = false,0`.

```
const $1 = {};  $1.int = 1;
const $2 = {};  $2.int = 2;
const $3 = {};  $3.int = $1.int + $2.int;
```

(a) Before optimization: 3 objects.

```
const $1 = 1;
const $2 = 2;
const $3 = $1 + $2;
```

(b) After optimization: 0 objects.

Fig. 26. Simplified JavaScript code for $1 + 2$.

unnecessary in this case. As shown in Fig. 26b, we do not need to create any objects when compiling $1 + 2$. Instead of $\{\texttt{int} \Rrightarrow 1\}$, we can directly use 1 if it is not part of a merge. This is a significant optimization for programs that heavily rely on arithmetic. In addition to integers, we also optimize the compilation for floating-point numbers, boolean values, and strings in a similar way.

The optimization becomes a bit more complicated when interacting with parametric polymorphism, as we cannot statically know the type of a polymorphic term. Therefore, to guarantee a consistent runtime representation of primitive values, we also add runtime primitiveness checks and perform the optimization to those polymorphic terms whose types are instantiated primitive. For an artificial example, consider the following CP code:

```
idPartly A B (x: A&B): B = x;
idPartly @String @Int        ("foo" , 48)         --> 48
idPartly @String @(Int&Bool) ("foo" , 48 , true)  --> 48 , true
```

We know that the return type of `idPartly` is `B`, but we cannot statically know if `B` is a primitive type. We should do the optimization in the first application above but not in the second one. Such a decision is made at run time by checking `B`'s primitiveness. By this means, we can make sure that the runtime representation of primitive values is consistent and efficient.

6.5 Selected Rules for Destination-Passing Style

To help to understand the implementation of destination-passing style, we present the compilation process in the form of type-directed rules. The full set of rules can be found in Appendix A. We select a few representative ones here. Besides destination-passing style, the design of the compilation rules is greatly influenced by the optimization that reduces object copying (discussed in Section 6.4). We will revisit the examples in Fig. 23, Fig. 24, and Fig. 25 to show how the optimized code is generated systematically.

In the compilation rules, we have three kinds of destinations:

- z stands for a non-empty destination, where the context is a merge. We will store the current result as a field in the destination object z.
- $y?$ stands for an optional destination, where the context is a function body. Since the last parameter of a compiled function can be **`undefined`**, we do not statically know if the destination is present.
- **nil** stands for no destination, which means that the context is neither a function body nor a merge.

An alternative approach that avoids the case of optional destinations is to compile each function twice: one with a destination and the other without. This allows the appropriate version to be chosen statically. We leave the exploration of this variant for future work.

Seven rules for type-directed compilation are selected in Fig. 27a. A rule $(\Gamma; dst \vdash e \Leftrightarrow A \rightsquigarrow \mathtt{J} \mid z)$ basically reads as: given a typing context Γ and a destination dst, the F_i^+ term e is checked/inferred to have type A and is compiled to variable z in JavaScript code J. That is, after running J, the result is stored in the JavaScript variable z. Let us take the variable access in F_i^+ as an example. We perform case analysis on the destination: if the destination is present, we copy the contents of x to z (J-Var); if the destination is absent, we directly return the variable x (J-VarNil); if the destination is optional, we dynamically check the presence of y and copy the contents of x only if y is present (J-VarOpt). Since the destination is set optional for the function body (J-Abs), J-VarOpt is used instead of the other two if the function body is a variable access. This is how we get the optimized code for an identity function in Fig. 23b. As for the function presented in Fig. 25, the body is a merge of two literals. There is only one version of J-Merge, which assumes that a destination is provided, so a bridge rule J-Opt is used to properly set the destination. Subsequently, J-Merge delegates the compilation to the two subterms (e.g. J-Int) and concatenates the JavaScript code.

Concerning function application, we select three rules in Fig. 27b. A rule $(\Gamma; dst \vdash x : A \bullet y : B \rightsquigarrow \mathtt{J} \mid z : C)$ basically reads as: given a typing context Γ and a destination dst, applying the compiled function in x of type A to the compiled argument y of type B yields variable z of type

Destinations $dst ::= \textbf{nil} \mid y? \mid z$

JavaScript code $\mathsf{J} ::= \varnothing \mid \mathsf{J}_1; \mathsf{J}_2 \mid$ `code`

$\boxed{\Gamma; dst \vdash e \Leftrightarrow A \rightsquigarrow \mathsf{J} \mid z}$ *(Type-directed compilation)*

J-Var
$$\frac{x : A \in \Gamma}{\Gamma; z \vdash x \Rightarrow A \rightsquigarrow \texttt{copy(z, x);} \mid z}$$

J-VarOpt
$$\frac{x : A \in \Gamma}{\Gamma; y? \vdash x \Rightarrow A \rightsquigarrow \texttt{if (y) copy(y, x);} \mid x}$$

J-VarNil
$$\frac{x : A \in \Gamma}{\Gamma; \textbf{nil} \vdash x \Rightarrow A \rightsquigarrow \varnothing \mid x}$$

J-Abs
$$\frac{\Gamma, x : A; y? \vdash e \Leftarrow B \rightsquigarrow \mathsf{J} \mid y_0}{\Gamma; z \vdash \lambda x{:}A.\ e{:}B \Rightarrow A \to B \rightsquigarrow \texttt{z["func\_|B|"] = (x, y) => \{ J; return y0; \};} \mid z}$$

J-Opt
$$\frac{\Gamma; z \vdash e \Leftrightarrow A \rightsquigarrow \mathsf{J} \mid z}{\Gamma; y? \vdash e \Leftrightarrow A \rightsquigarrow \texttt{var z = y || \{\}; J;} \mid z}$$

J-Merge
$$\frac{\Gamma; z \vdash e_1 \Rightarrow A \rightsquigarrow \mathsf{J}_1 \mid z \qquad \Gamma; z \vdash e_2 \Rightarrow B \rightsquigarrow \mathsf{J}_2 \mid z \qquad \Gamma \vdash A * B}{\Gamma; z \vdash e_1 \,,\, e_2 \Rightarrow A \,\&\, B \rightsquigarrow \mathsf{J}_1; \mathsf{J}_2 \mid z}$$

J-Int
$$\Gamma; z \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \texttt{z.int = n;} \mid z$$

(a) Type-directed compilation.

$\boxed{\Gamma; dst \vdash x : A \bullet y : B \rightsquigarrow \mathsf{J} \mid z : C}$ *(Function application)*

JA-ArrowEquiv
$$\frac{A \fallingdotseq C}{\Gamma; z \vdash x : A \to B \bullet y : C \rightsquigarrow \texttt{x["func\_|B|"](y, z);} \mid z : B}$$

JA-ArrowOpt
$$\frac{A \fallingdotseq C}{\Gamma; z_0? \vdash x : A \to B \bullet y : C \rightsquigarrow \texttt{var z = x["func\_|B|"](y, z0);} \mid z : B}$$

JA-ArrowNil
$$\frac{A \fallingdotseq C}{\Gamma; \textbf{nil} \vdash x : A \to B \bullet y : C \rightsquigarrow \texttt{var z = x["func\_|B|"](y);} \mid z : B}$$

(b) Function application.

Fig. 27. Selected rules for destination-passing style.

C in JavaScript code J. All three rules deal with the simple cases where the parameter type is equivalent to the argument type, so we do not need to insert any coercion for the argument. Again, we perform case analysis on the destination (JA-ArrowEquiv, JA-ArrowNil, and JA-ArrowOpt). This explains why we get different JavaScript code for the two function calls in Fig. 24.

6.6 Separate Compilation

Lastly, our implementation supports separate compilation. This is usually difficult to achieve in a programming language with a high level of extensibility and modularity that can solve the expression problem (CP's solution is covered in Section 3.4). The difficulty of separate compilation

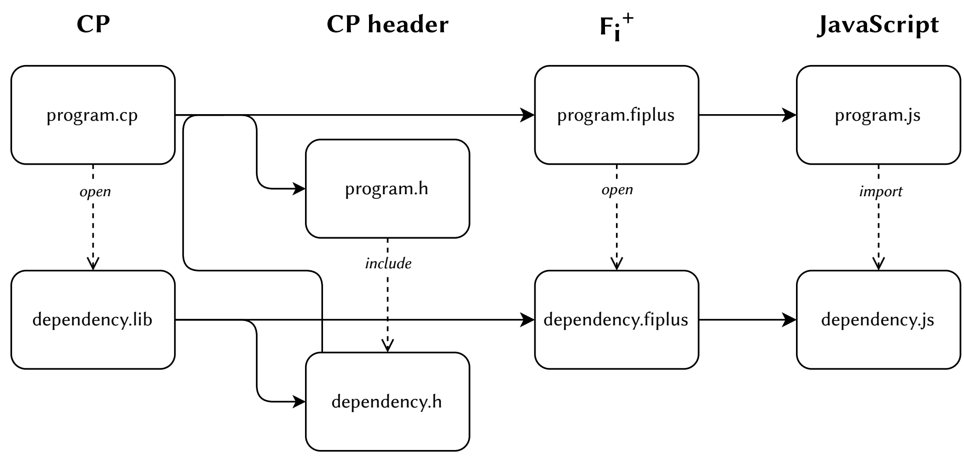


Fig. 28. A flowchart of separate compilation in CP.

```
-- example.cp
open strings;
type Rcd = { m: String; n: String };
mkA = trait [this: Rcd] ⇒ {
  m = "foobar";
  n = toUpperCase this.m;
};
```

```
-- example.cp.h
include "strings.lib.h";
type Rcd = { m: String; n: String };
term mkA : Trait<{ m: String }&{ n: String }
    ⇒ { m: String }&{ n: String }>;
```

Fig. 29. A CP file and its corresponding header file.

in the presence of modularity has been previously studied in the context of *feature-oriented programming* [Apel and Kästner 2009; Prehofer 1997]. As identified by Kästner et al. [2011], modularity can be divided into two categories: *cohesion* and *information hiding*. The cohesive approach often employs source-to-source transformations, which require the whole source code to be available. As a result, they achieve modularity at the cost of modular type checking and separate compilation. Many existing feature-oriented tools, such as AHEAD [Batory et al. 2004], FeatureC++ [Apel et al. 2005], and FeatureHouse [Apel et al. 2013b], fall into this category. The second approach is based on strong interfaces and information hiding. This notion of modularity is underrepresented in feature-oriented software development, but it is emphasized in the community of programming languages and is employed by `gbeta` [Ernst 2000] and CP.

Both `gbeta` and CP can solve the expression problem via family polymorphism [Ernst 2004] without sacrificing separate compilation. However, separate compilation affects the performance of attribute lookup in `gbeta`. Since an object in `gbeta` may have more mixin instances at run time than what is statically known, and the mixin instances may occur in a different order, the offset of an attribute cannot be determined statically. Especially when separate compilation is desired, we cannot do whole program analysis to optimize attribute lookup for some specific inherited classes. As a result, `gbeta` has to perform a linear search through super-mixins to look up inherited attributes. In contrast, attribute lookup in CP, even with dynamic inheritance, is much more efficient, and no linear search is needed.

The flowchart in Fig. 28 gives an overview of the compilation process from CP all the way down to JavaScript, which is almost a textbook example of separate compilation. Like most programming languages, the compilation unit of CP is a file. One can refer to another file using `open` directives in CP, which compile to JavaScript module `import` statements. To provide sufficient type information for modular type checking, a CP file compiles to a JavaScript file as well as a CP header file. CP header files are similar to `.mli` files in OCaml and consist of type definitions, type signatures for terms, and references to other header files. See Fig. 29 for a slightly simplified example. Normally, header files are automatically generated by the CP compiler, but users can also edit them to hide some definitions that are supposed to be private. The compilation only depends on the file to be compiled and the headers files of its dependencies. As a result, compiling a file does not require recursively compiling its dependencies, and its dependents do not need recompilation as long as its header file is not changed (though its implementations may have changed).

Between CP and JavaScript, there is a core calculus F_i^+ [Bi et al. 2019; Fan et al. 2022]. Since our implementation of CP is based on the elaboration semantics formalized by Zhang et al. [2021], CP language constructs are first desugared into F_i^+ terms, and then these F_i^+ terms are compiled into JavaScript code. Both sets of elaboration rules are syntax-directed and compositional, and the elaboration contexts only include type information from header files in our implementation. That is why CP code can be separately compiled with the help of header files.

7 Empirical Evaluation

In this section, we conduct an empirical evaluation of the CP compiler. We analyze the impact of various optimizations in the CP compiler. Furthermore, we compare the efficiency of dynamic inheritance in CP with that in handwritten JavaScript code. The key takeaway from our empirical evaluation is that using a naive compilation scheme for merges can be orders of magnitude slower than optimized code. Our optimizations lead to code that can be competitive with similar handwritten JavaScript code. The benchmarks are available in the supplementary materials.

Experimental setup and benchmark programs. We performed experiments on a system featuring an Apple M1 Pro chip and 16GB RAM. JavaScript code was executed using Node.js 20.12.2 LTS. The outline of benchmark programs is presented in Table 1. The initial four benchmarks focus on general-purpose computations, while the latter four are adapted from recent work on compositional embeddings [Sun et al. 2022], showcasing CP's novel features. Among them, chart is the biggest program with around 300 lines of code. Challenges discussed in Section 3, including dynamic inheritance and family polymorphism, are prominent in the latter four benchmarks.

7.1 Ablation Study on Optimizations

The coercive subtyping semantics of CP raises important questions about efficiency since coercions have runtime costs and they are pervasively employed in generated code. There are essentially three main concerns that need to be addressed in obtaining an efficient compilation scheme for CP:

- **Efficient lookup.** Since merge lookup is pervasive, it is important to use a runtime representation for merges that enables efficient lookup.
- **Efficient merging and copying.** Since merging is frequent, it is important that the merging process is efficient and minimizes the amount of copying involved in merging.
- **Minimizing the cost of coercions.** Since our subtyping is coercive, it is fundamental that the cost of coercions is minimized. Furthermore, optimizations should avoid coercions when possible.

In our work, we have addressed the above three points. Our representation of merges as type-indexed records makes the cost of a merge lookup essentially the same as the cost of a JavaScript field

Table 1. Outline of the benchmark programs.†

fib	Calculating Fibonacci numbers without memoization.
fact	Some factorial functions multiplied together.
sieve	Sieve of Eratosthenes, an algorithm for finding prime numbers.
nbody	Numerical simulation of the n-body problem.
region	An embedded DSL for geometric regions.
chart	Generating SVG code for customizable charts.
fractal	Generating SVG code for a simple fractal called the Sierpiński carpet.
minipedia	Generating HTML code for a mini document with a computed table of contents.

† Some computations in the benchmark programs are repeated several times for longer and more stable execution time.

lookup, which is very efficient. This provides a major source of improvement over a representation with pairs, where lookup time can be linear. We believe that it is hard to do better in this dimension, at least if the goal is to target JavaScript. For merging, we rely on JavaScript's ability to copy object fields. An important concern for merging is to avoid the creation of intermediate objects, minimizing the amount of copying. The DPS optimization is particularly important for obtaining efficient merging. Like lookup, we believe that the CP compiler also achieves efficient merging. Finally, to mitigate coercions, we employ a hybrid model that combines inclusive and coercive subtyping. We only insert coercions when necessary and try to eliminate redundant coercions as much as possible. We have mentioned several optimizations in Section 6, two of which are avoiding unnecessary coercions: one for equivalent types and the other for record projections.

All the implemented optimizations should improve the performance of our CP compiler in theory. Here we select four representative ones to evaluate their impact in practice:

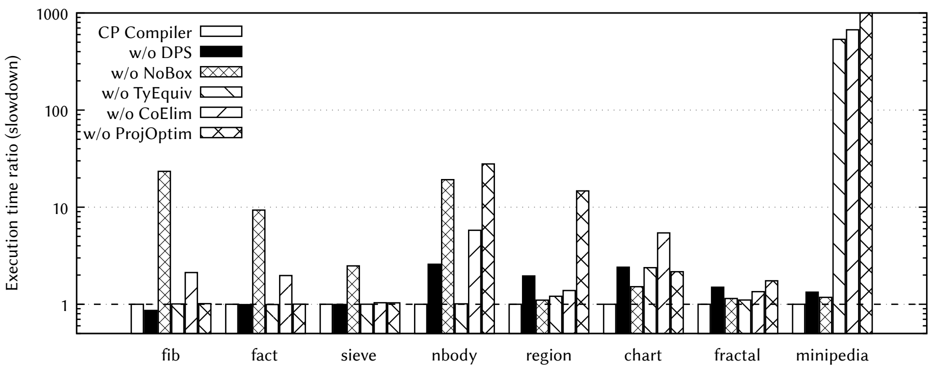
(1) Reducing intermediate objects using destination-passing style (DPS);
(2) Preventing primitive values from boxing/unboxing (NoBox);
(3) Eliminating coercions for subtyping between equivalent types (TyEquiv & CoElim);
(4) Avoiding the insertion of coercions for record projections (ProjOptim).

We conduct an *ablation study* on the four optimizations. Fig. 30b shows the execution time ratios (slowdowns) to the optimized JavaScript code when removing each optimization. Fig. 30a lists the original data for Fig. 30b in milliseconds. CP Compiler represents the most optimized version of the CP compiler, including all the aforementioned optimizations. The remaining variants are CP Compiler minus one optimization, including all other optimizations. To summarize the benchmark results, different optimizations show different degrees of speedup for different benchmarks, but we believe that the coercion-related ones are especially important when (dynamic) inheritance is concerned.

The first optimization (DPS) speeds up the execution of the latter five benchmarks by reducing the number of intermediate objects and object concatenations. In contrast, the former three benchmarks do not benefit from the optimization because they only perform arithmetic operations and no objects are involved. The first benchmark (fib) even becomes a bit slower because the optimization inserts extra checks into function bodies to test if destination objects are present. Overall, the speedup ratios are 2.6× at most. This optimization clearly helps for programs involving objects and merging, although the benefits of this optimization are smaller than optimizations on coercions. In

	fib	fact	sieve	nbody	region	chart	fractal	minipedia
CP Compiler	2837	1433	1736	1704	1944	516	4578	45
w/o DPS	2451	1422	1728	4402	3806	1243	6861	60
w/o NoBox	66348	13369	4314	32716	2144	783	5249	53
w/o TyEquiv	2860	1425	1738	1722	2349	1229	5064	24080
w/o CoElim	6020	2832	1801	9851	2693	2803	6173	30192
w/o ProjOptim	2880	1438	1795	47505	28591	1117	7990	OOM‡

(a) Execution time (ms) of the JavaScript code generated by variants of the CP compiler.



(b) Execution time ratios (slowdowns) of different variants to the optimized JavaScript code.‡

‡ The bar that exceeds the frame represents *JavaScript heap out of memory* (OOM) for minipedia w/o ProjOptim.

Fig. 30. Ablation study on optimizations for the CP compiler.

essence, intermediate objects are not the main bottleneck in the JavaScript code generated by the CP compiler, although they still have a considerable cost for many programs.

The second optimization (NoBox) is important for primitive operations such as arithmetic, which complements the first optimization. It speeds up all benchmarks since primitive operations are inevitable in practical programs. It brings around 23× speedup for fib and around 19× speedup for nbody because they involve a lot of arithmetic operations. Numbers do not need to be boxed/unboxed in the optimized JavaScript code, so the performance is improved significantly.

The analysis for the third optimization is split into two parts for a finer-grained analysis. We have a version of the CP compiler that only removes coercions for *syntactically equal* types but does not eliminate other coercions for *equivalent* types (w/o TyEquiv). The other version does not eliminate redundant coercions at all (w/o CoElim). Some benchmarks (such as chart and minipedia) make use of equivalent types a lot, hence their performance is already affected by removing TyEquiv. After further removing CoElim, most benchmarks experience significant slowdowns (up to 671× slower in the worst case for minipedia).

The last optimization (ProjOptim) targets coercions for record projections, so the benchmarks that do not use records (such as fib, fact, and sieve) are not affected at all. Among the relevant benchmarks, nbody becomes around 28× slower without this optimization. This is because the masses, velocities, and coordinates of the bodies are all stored in records. Note that the JavaScript code generated for minipedia runs out of memory, so there is no data in Fig. 30a, and the exception is represented by a bar that exceeds the frame in Fig. 30b.

In conclusion, all optimizations work in practice. The elimination of redundant coercions has a particularly significant impact on the performance. The representation of JavaScript objects (or extensible records in general) brings forth a class of equivalent types, whose terms share the same shape. We can then avoid the coercions between these types but still obtain an equivalent object as a result. This optimization has a significant impact but cannot be done in previous work [Dunfield 2014; Oliveira et al. 2016] because they use nested pairs as the elaboration target of merges. Since pairs are order-sensitive, they require coercions that can be avoided with order-insensitive objects (see also the discussion in Section 4.1). Together with the faster lookup by type indices in objects, the JavaScript code generated by the CP compiler achieves reasonable performance.

7.2 Comparison with Handwritten JavaScript Code

Our focus in this paper is on the type-safe compilation of dynamic inheritance and the efficient compilation of languages with merges. A first natural question to ask is how the new compilation scheme compares against existing compilation schemes for merges. Unfortunately, such a direct comparison is not feasible for a few different reasons. Firstly, the only other compiler for a language with merges is Stardust by Dunfield [2014]. However, Stardust targets ML, instead of JavaScript. Thus, a direct comparison of performance would not be possible. Furthermore, Stardust does not support distributive subtyping and nested composition. Thus, most of our examples and case studies cannot be encoded in Stardust. Nevertheless, in Section 4.1, we have highlighted some advantages of using our record-based representation versus using pairs (which Stardust employs) in the compilation of merges.

In spite of the above-mentioned difficulties of a direct comparison, it is still helpful to do an elementary quantitative analysis with handwritten JavaScript code to assess the impact of the coercive semantics of CP. Although we have worked hard to eliminate unnecessary coercions, the JavaScript code generated by the CP compiler still includes plenty of coercions. In contrast, handwritten JavaScript code is coercion-free, and subtyping in TypeScript has no cost. It would be unrealistic to expect a stable performance that is competitive with JavaScript, especially since our implementation is still a proof of concept for our compilation scheme. However, ideally, the performance penalty imposed by coercions should not be too high.

A brief comparison is made based on the former four benchmarks, namely fib, fact, sieve, and nbody (we will explain region^0 later). Fig. 31c shows the execution time ratios (slowdowns) of the JavaScript code generated by the CP compiler compared to the handwritten JavaScript code, and Fig. 31a lists the original data. They mainly demonstrate general-purpose computations. The handwritten JavaScript code is transliterated from the corresponding CP code in order to make an apples-to-apples comparison. It follows a functional programming style similar to CP and may not be idiomatic in JavaScript. The performance of the JavaScript code generated by the CP compiler is slightly slower than that of the handwritten code for fib, fact, and sieve. The biggest slowdown is around 3× for nbody, partly because the manipulations of records and arrays in CP are less efficient than in native JavaScript. Moreover, our treatment of **`let`** expressions is oversimplified. In CP, `let x = e1 in e2` is desugared into `(\x → e2) e1`, which is much slower than **`const`** statements in JavaScript. In nbody, there are several nested **`let`**s in recursive functions, introducing significant overhead.

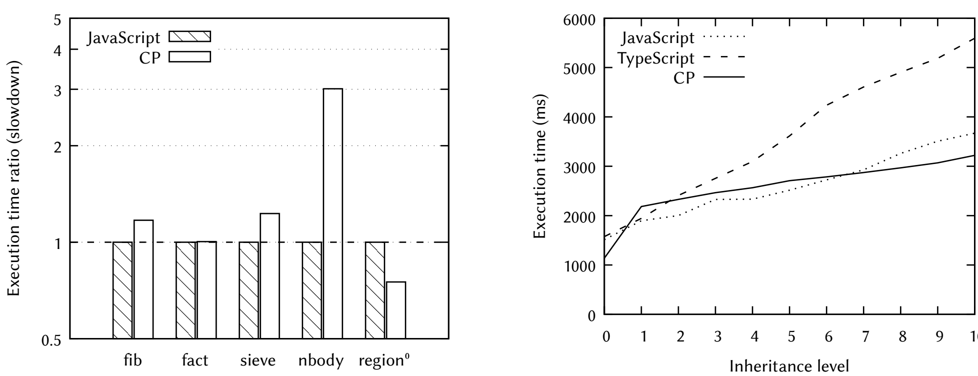
The latter four benchmark programs make use of CP's novel features, making transliteration to JavaScript difficult. Nevertheless, we adapt the fifth benchmark (region) to make a comparison between conceptually equivalent programs. To recap, the benchmark program is mainly an embedded DSL for geometric regions [Hudak 1998]. For modular extension, the DSL is implemented with techniques of family polymorphism, which are described in Section 2.4 for JavaScript and in Section 3.4 for CP. Both implementations heavily rely on class/trait inheritance, so the performance penalty of inheritance is well demonstrated in this benchmark. Furthermore, we change the number of inheritance levels from 0 to 10 (region^n represents that the desired method is in the n-level

	fib	fact	sieve	nbody	region^0
JavaScript	2423	1427	1413	566	1513
CP	2837	1433	1736	1704	1137

(a) Execution time (ms) for five benchmarks.

Inheritance level	0	1	2	3	4	5	6	7	8	9	10
JavaScript	1513	1896	2002	2328	2333	2515	2724	2928	3260	3507	3670
TypeScript	1575	1944	2409	2755	3096	3614	4236	4606	4903	5186	5593
CP	1137	2184	2329	2465	2565	2708	2785	2873	2968	3069	3221

(b) Execution time (ms) for region$^{0..10}$.



(c) Bar chart for five benchmarks.

(d) Line chart for region$^{0..10}$.

Fig. 31. Comparison between JavaScript code generated by the CP compiler and handwritten code.

super-trait/-class) to see the trend of the performance penalty. In other words, region^0 is *monolithic* code with a single trait/class and no inheritance hierarchy. At level one, we introduce a slightly more modular version of region with one level of inheritance: there is a super-trait/-class and a sub-trait/-class. Higher levels simply introduce more inheritance layers. The results are shown in Fig. 31b and Fig. 31d.

Besides CP and JavaScript, a TypeScript version is also included for this comparison. The source code is simply the JavaScript version plus type annotations. We use the official TypeScript compiler to compile it to JavaScript and then use Node.js to execute the JavaScript code. The TypeScript code has a different performance profile from the JavaScript code because the TypeScript compiler by default (as of the current version 5.4) desugars classes into prototypes. This is due to the default compilation target being ECMAScript 3 [ECMA 1999] for best compatibility, which does not support classes. Newer versions of Node.js (based on ECMAScript 6 [ECMA 2015] or above) natively support classes, so the handwritten JavaScript directly uses classes. To sum up, the difference between JavaScript and TypeScript in the benchmark is mainly classes versus prototypes.

Without inheritance (region^0), the JavaScript code generated by the CP compiler is *faster* than the handwritten JavaScript and TypeScript code. This is because the technique of nested anonymous classes is neither idiomatic nor efficient in JavaScript. In contrast, nested traits themselves do not introduce extra runtime overhead in CP. However, when the desired method is one level up in the inheritance hierarchy, the CP compiler generates around 2× slower code, compared to the monolithic version, because coercions are inserted for nested trait composition. For the monolithic version, there are almost no coercions in the CP code. Then the performance penalty increases more smoothly when the inheritance level is higher. The CP compiler regains its leading position when the number of inheritance levels is higher than 6. In contrast, TypeScript has the steepest curve. The desugared prototype-based code generated by the TypeScript compiler is the least efficient among the three implementations.

In conclusion, the performance penalty of coercions brought by trait inheritance is not negligible but increases more smoothly with the number of inheritance levels than in handwritten JavaScript. This is partly due to the efficient lookup by type indices in extensible records (or, more specifically, JavaScript objects). Looking up a deeply nested method in the inheritance hierarchy can be slow in JavaScript, but this is not the case in CP.

8 Related Work

Compilation of inheritance. In his excellent survey on inheritance, Taivalsaari [1996] distinguishes two strategies for implementing inheritance: *delegation* and *concatenation*. Most prototype-based languages, such as Self [Ungar and Smith 1987] and JavaScript, implement inheritance via delegation, where an object contains a reference to its prototype (e.g. `__proto__` in JavaScript), and methods that are not found in the current object will be delegated to its ancestors in the prototype chain. In contrast, CP implements inheritance via concatenation (a.k.a. *merging* throughout the paper), where a trait is self-contained and itself contains all the methods of its ancestors. Although some copying is involved, the concatenation strategy is more efficient than delegation in terms of method lookup.

To improve the performance of method lookup, newer implementations of the Self language cache all lookup results for a polymorphic call site in a *polymorphic inline cache* (PIC) [Hölzle et al. 1991]. The methods cached in a PIC will be inlined into the caller to further reduce the overhead of method calls. Since a PIC is empty until a method is called for the first time, dynamic recompilation is required to optimize the code at run time. Moreover, the presence of *dynamic inheritance* may lead to a full method lookup in Self [Chambers 1992]. Modern JavaScript engines, such as V8 used in Node.js, utilize similar PIC-based techniques to optimize method calls. Though the CP compiler does not implement inlining at all, which is definitely a useful optimization, it is still efficient in terms of method lookup, and dynamic inheritance *never* causes a slower lookup.

Typical compilers for mainstream class-based languages, such as C++ and Java, add a *virtual method table* (vtable) [Driesen et al. 1995] to each object to avoid searching for methods in the inheritance hierarchy at run time. A vtable is basically an array of function pointers, associating each method name (and parameter types if overloaded) with its implementation. Similarly, CP compiles an object to a type-indexed record, which also associates each method name and type with the corresponding implementation, among other fields. What is more, CP allows for first-class classes (traits) and dynamic inheritance, which are not supported by most mainstream languages. This is one of the key differences of our work compared to other OOP language compilers.

Another significant difference from mainstream OOP languages is that our compilation of inheritance is based on the denotational model by Cook and Palsberg [1989]. In this model, classes (traits) are encoded as functions, and inheritance is essentially merging functions, which is illustrated in Section 3.2. That is why the source language of the compilation scheme (λ_i^+) does not contain any notion of classes or objects. Such encodings are common in the literature on foundations for statically typed OOP [Bruce 2002; Bruce et al. 1999; Pierce 2002], and they largely simplify the formalization of compilation and its metatheory.

Multiple inheritance is a well-known troublemaker in OOP languages, bringing the diamond problem and method conflicts, among other issues. Alternative notions like *mixins* [Bracha and Cook 1990] and *traits* [Ducasse et al. 2006] are proposed to alleviate the issues. A core difference between mixins and traits is how they handle conflicts when the same method name occurs in multiple ancestors. Mixins resolve conflicts implicitly by linearization (e.g. C3 linearization [Barrett et al. 1996]). However, the implicit resolution of conflicts may conceal accidental conflicts and lead to subtle bugs. Traits, on the other hand, require the programmer to resolve conflicts explicitly. CP adopts the trait model and imposes the disjointness constraint on merging (and trait inheritance). Note that the disjointness constraint does not only consider the method names but also takes into account the types of the methods, so the methods with the same name but different return types are considered disjoint and do not conflict with each other. By this means, CP tries to reach a balance between safety and flexibility.

Dynamic inheritance and first-class classes. While various forms of multiple inheritance are well studied and implemented in some popular languages, such as C++, Ruby, and Scala, dynamic inheritance is more challenging and involved, especially in terms of static typing. In the literature of OOP, dynamic inheritance is often discussed in a broader context of *first-class classes* [Strickland et al. 2013], where inherited classes can be determined at run time, among other dynamic features. There are only a few statically typed languages that support first-class classes. To the best of our knowledge, they are `gbeta` [Ernst 2000], TypeScript [Microsoft 2012], Typed Racket [Takikawa et al. 2012], Wyvern [Lee et al. 2015], and most recently, CP [Zhang et al. 2021]. As elaborated in Section 2, the most popular one, TypeScript, has significant type-safety issues when dealing with dynamic inheritance.

Typed Racket is gradually typed and uses row polymorphism to represent class types. Similarly to the disjointness constraints in CP, there are constraints on row variables to express absence, and thus the *inexact superclass problem* that TypeScript suffers from is resolved in Typed Racket. However, the absence constraint on a row variable only includes the method name but not the type, so the dynamically inherited class is more restricted than in CP. Moreover, Xie et al. [2020] formally prove that CP's disjoint polymorphism is more powerful than similar forms of row polymorphism. Furthermore, unlike Typed Racket, CP can model virtual classes and family polymorphism.

Wyvern is a language for design-driven assurance, and Lee et al. [2015] explored a foundational account of first-class classes based on tagging [Glew 1999]. Similarly to our formalization, they give an elaboration semantics of an OOP language. However, their theory is very different from ours, and they target a more sophisticated calculus with hierarchical tagging and dependent types. In contrast, our target language is a standard record calculus. Furthermore, their calculus cannot model multiple inheritance or family polymorphism, and their implementation is an interpreter rather than a compiler.

The `gbeta` language is the most interesting one and is the closest to CP because it supports dynamic multiple inheritance and family polymorphism. However, separate compilation was not supported at the time when Ernst [2000] wrote his dissertation because of some technical issues with the Mjølner BETA persistence support. If this factor is disregarded, separate compilation can still be accomplished, but at the cost of efficient attribute lookup.[8] Since an object in `gbeta` may have more mixin instances at run time than what is statically known, and the mixin instances may occur in a different order, the offset of an attribute cannot be determined statically. As a result, `gbeta` has to perform a linear search through super-mixins to look up inherited attributes. In contrast, attribute lookup in CP, even with dynamic inheritance, is more efficient, and no linear search is needed.

The notion of *patterns* in `gbeta` unifies classes and methods, and patterns can be composed using the combination operator '&', which is similar to the merge operator ',' in CP. Though dynamic

[8]In email communications, Erik Ernst, the author of `gbeta`, confirmed having an implementation with separate compilation but linear-time lookup of attributes.

multiple inheritance can be achieved using '&', only a subset of dynamic combinations is safe, where at least one of the classes being composed must be created by single inheritance [Ernst 2002]. Otherwise, the C3 linearization algorithm used by `gbeta` may fail at run time. In contrast, dynamic inheritance via '$,$' is completely type-safe, because CP utilizes disjointness to avoid conflicts, and no linearization is needed. This difference has been summarized earlier as *mixins versus traits*. Some other notable consequences of this difference are:

- '$,$' is commutative, while '&' is not;
- '$,$' supports mutual dependencies between traits, while '&' rejects such cycles.

Virtual classes and family polymorphism. Virtual classes [Madsen and Møller-Pedersen 1989], similarly to virtual methods, are nested classes that can be overridden in subclasses. Virtual classes enable family polymorphism [Ernst 2001], which can naturally solve the expression problem [Ernst 2004]. The idea of virtual classes was initially introduced in the BETA programming language [Madsen et al. 1993] and later generalized in `gbeta` [Ernst 2000]. CaesarJ [Aracic et al. 2006], an aspect-oriented programming language based on Java, also supports virtual classes but does not allow cross-family inheritance and dynamic inheritance. Newspeak [Bracha et al. 2010], a descendant of Smalltalk, combines virtual classes and first-class modules (i.e. instances of top-level classes) but is dynamically typed. The calculi Jx [Nystrom et al. 2004], J& [Nystrom et al. 2006], *vc* [Ernst et al. 2006], Tribe [Clarke et al. 2007], and .FJ [Saito et al. 2008], just to name a few, formalize virtual classes with static inheritance but do not support dynamic inheritance.

Zhang and Myers [2017] propose the Familia programming language that unites object-oriented polymorphism and parametric polymorphism by unifying interfaces and type classes. In Familia, a mechanism of family polymorphism based on *nested inheritance*, similarly to Jx [Nystrom et al. 2004], is also deployed. During compilation, a *linkage* is computed for every class, which consists of a self-reference, a dispatch table, and the linkages of its nested classes, among others. At the heart of the mechanism is *further binding* [Madsen et al. 1993]: rewiring self-references for nested classes. Further binding is realized in Familia by linkage concatenation between families. This process is similar to the nested trait composition in CP, but there is a significant distinction in terms of separate compilation. CP *only* needs type information of the imported modules at compile time, while Familia requires class linkages that contain some implementation details of the imported modules (e.g. method definitions) and copy these details from superclasses' linkages. In this sense, with linkages, Familia supports some degree of separate compilation, but not to the same extent as the CP compiler does. Moreover, since Familia does not support dynamic inheritance, their class hierarchies are determined statically. In contrast, CP supports dynamic trait composition, which brings extra flexibility.

More recently, Kravchuk-Kirilyuk et al. [2024] propose Persimmon, a functional programming language that features extensible variant types and extensible pattern matching. CP also supports them via compositional interfaces and method patterns. Persimmon additionally allows types to be members of a family, relying on the support for *relative path types* [Saito et al. 2008] in their core calculus. Internally, Persimmon makes use of *linkages* that are similar to those in Familia. An important limitation of their current design is that modular type checking and separate compilation are *not* supported for multi-file programs, while CP fully supports them.

Elaboration of intersection types and the merge operator. Dunfield [2014] shows that *unrestricted* intersection types and a term-level merge operator [Reynolds 1997] can encode various features like overloading and multi-field records, and they can be elaborated into product types and pairs. However, her approach lacks the critical property of coherence, i.e. the property that ensures the result of a merge is unambiguous. In the follow-up work on *disjoint* intersection types [Oliveira et al.

2016], the merged components are required to be disjoint with each other to avoid the semantic ambiguity. Alpuim et al. [2017] added parametric polymorphism to the calculus. Bi et al. [2018, 2019] further enhanced the intersection subtyping with distributivity, enabling more novel features like nested composition and family polymorphism. In other words, only Bi et al.'s F_i^+ calculus fully covers the topics mentioned in Section 3. All the aforementioned work employs elaboration semantics with standard λ-calculi serving as targets. They use nested pairs as the target of elaboration, and consequently, the time complexity of extracting a component by type can degenerate to linear in the worst case. In addition, extensible records require fewer coercions than nested pairs because some different source terms compile to equivalent records. These differences from our CP compiler have been discussed in detail in Section 4.1. In short, they do not consider more efficient runtime representations or eliminating redundant coercions, nor do they have benchmarks to evaluate performance. Instead, their focus is on proving the type safety and coherence of the elaboration. Furthermore, none of the aforementioned work develops a language with separate compilation units.

The compilation of merges in our work has similarities to the compilation of *type-indexed rows* [Shields and Meijer 2001], where record labels are discarded and record fields are sorted by their types. However, the work on type-indexed rows does not consider subtyping, which eliminates many of the issues that we had to deal with. For instance, they do not need to apply coercions to ensure that information statically hidden by subtyping is also hidden at run time.

Compilation of extensible records. Ohori [1995] investigates a polymorphic record calculus and introduces an efficient type-directed compilation method for records. Following the type-inference stage, records are converted into vectors with explicit indexing. However, his records are not extensible, and his method has difficulties to handle subtyping. Subtyping for records frequently enables field hiding and reordering, rendering it impossible to determine a label's offset statically. Gaster and Jones [1996] propose a compilation technique for polymorphic extensible records that utilizes qualified types [Jones 1994]. During the compilation process to the target language, supplementary parameters are introduced to determine suitable offsets. This approach is integrated into Hugs, a well-known implementation of Haskell, as an extension. Their system is later generalized by type-indexed rows [Shields and Meijer 2001]. In summary, subtyping and record concatenation (or merges) pose significant challenges to the compilation of extensible records. Our work takes pragmatic considerations into account, including targeting widely used dynamic languages such as JavaScript. As a result, we rely on the primitive support of objects and object extension in our target language and do not delve into low-level representations of extensible records, for which a comprehensive summary can be found in the paper by Leijen [2005].

Compilation of feature-oriented programming. Feature-oriented programming (FOP) [Apel and Kästner 2009; Prehofer 1997] is a programming paradigm that aims to modularize features in software product lines [Apel et al. 2013a]. There is a debate on what modularity exactly means, and Kästner et al. [2011] mention two notions of modularity: *cohesion* and *information hiding*. The majority of FOP work [Apel et al. 2013b, 2005; Batory et al. 2004] focuses on the notion of cohesion and basically does source-to-source transformations, which hinders modular type checking and separate compilation. There is some other work, such as Jiazzi [McDirmid et al. 2001] and Scala [Odersky and Zenger 2005], leveraging information hiding instead and supporting modular type checking and separate compilation. However, FOP is usually achieved via verbose design patterns or metaprogramming in those languages. For instance, some precursor work of compositional programming, done in Scala, employs design patterns based on object algebras [Oliveira and Cook 2012] to achieve FOP [Oliveira et al. 2013; Rendel et al. 2014]. Since merging is not directly supported in Scala, specialized composition operators are required to simulate merges for different object algebra interfaces. The creation and use of those composition operators cause a significant burden for developing programs. In contrast, CP natively supports merges, eliminating the need for such specialized composition operators and supporting FOP more directly.

Delta-oriented programming (DOP) [Schaefer et al. 2010] is an extension of FOP, which features delta modules that can add, remove, or modify classes. A feature module is a delta module without the remove operation. DOP supports compositional type checking [Bettini et al. 2013] at the level of source code. More recently, a core calculus for *dynamic* DOP is proposed by Damiani et al. [2018] to support runtime variability [Hallsteinsen et al. 2008] and is proven to be type-safe. However, dynamic DOP is not yet implemented, and its separation compilation is unexplored.

9 Conclusion

CP is unique in that it supports dynamic inheritance, multiple inheritance, and family polymorphism all together in a type-safe manner. This paper proposes an efficient compilation scheme for CP, which features modular type checking and separate compilation. Not only have we presented formalized rules that capture the main ideas of compiling merges to type-indexed records, but we also provide a concrete implementation that targets JavaScript. In addition, benchmarks are included to evaluate our CP compiler empirically. The experimental results validate that our compilation scheme and optimizations lead to reasonable efficiency of generated code. More importantly, the type safety of our compilation scheme has been mechanically proven in Coq. We hope our work will benefit future work on type-safe compilation for dynamic inheritance or family polymorphism.

Future work. An obvious direction for future research is to formalize the compilation of parametric polymorphism. This endeavor would require significant effort, because not only will type variables and disjoint quantification complicate the metatheory, but the target calculus also needs to support first-class labels. A second direction is to prove coherence. While we have briefly sketched how a proof of coherence of the elaboration could be done, by adapting ideas in previous work [Bi et al. 2018], we have not done this proof. Thus, completing this proof would be interesting, although we believe that our focus on generating efficient coercions may add significant complexity to this proof. A possibility here is to define a simplified elaboration semantics targeting a language with records, but not aimed at optimizing coercions. This should be helpful for simplifying a coherence proof. Another interesting result would be to give a direct semantics for λ_i^+ and show that the elaborated terms in the target preserve the semantics of the source term.

In our elaboration rules, all top-like terms are treated as $\top$ and elaborated to an empty record (see rules Ela-Top, Ela-TopAbs, and Ela-TopRcd). Moreover, the coercive subtyping rule S-Top coerce a target term to an empty record if B is top-like in $A <: B$. As a result, side effects in top-like terms are erased during elaboration, which is not desired in imperative languages. For example, $(\lambda r.\ r := 1) : \mathbf{Ref}\ \mathbb{Z} \to \top$ is elaborated to $\{\}$, and the original function is erased. One potential solution is not to erase the top-like terms during the elaboration. For example, we can elaborate the previous expression to $\{|\mathbf{Ref}\ \mathbb{Z} \to \top| \mapsto \lambda r.\ \epsilon\}$, assuming $r := 1$ is elaborated to ϵ. However, this change breaks the current design of equivalent types because top-like terms can have different representations now. Sun [2025] briefly discusses this issue in the section of future work.

Concerning implementation, our prototype of the CP compiler is not as fully fledged as existing compilers for other functional languages. In addition, more optimizations should be done to improve the performance of the generated JavaScript code, especially on coercions. For example, when compiling an upcast $(48 \,,\, \mathbf{true}) : \mathbf{Int}$, we could use masks to hide the boolean part instead of deleting that field. While this paper focuses the empirical evaluation on the performance of the generated JavaScript code, the compilation time is also an important factor in practice. For instance, type splitting (rule S-Split) used in our subtyping algorithm is not efficient enough and can lead to a significant slowdown when complex intersection types are involved. It is worthwhile investigating a more efficient algorithm for distributive subtyping with intersection types in the future.

A Compilation Scheme from F_i^+ to JavaScript

The syntax of F_i^+ is defined below, where the main differences from λ_i^+ are the addition of parametric polymorphism and fixpoint expressions:

$$\begin{array}{ll} \text{Types} & A, B ::= \top \mid \bot \mid \mathbb{Z} \mid X \mid A \to B \mid \forall X{*}A.\ B \mid \{\ell : A\} \mid A \,\&\, B \\ \text{Expressions} & e ::= \{\} \mid n \mid x \mid \mathbf{fix}\, x{:}A.\ e \mid \lambda x{:}A.\ e{:}B \mid e_1\ e_2 \mid \Lambda X{*}A.\ e{:}B \mid e\,A \mid \{\ell = e\} \mid e.\ell \mid e_1 \,{,}\!\!,\, e_2 \mid e : A \end{array}$$

The compilation scheme we describe here directly generates JavaScript code instead of λ_r terms, which is closer to the actual implementation. We denote the generated JavaScript code by $\mathcal{J}$, which can be empty ($\varnothing$), concatenation of two pieces of code ($\mathcal{J}_1; \mathcal{J}_2$), or some predefined code that is listed in Fig. 32. There are some notations for type indices, which are actually implemented as strings in JavaScript (as discussed in Section 4.5). Destinations [Shaikhha et al. 2017] also play an important role in the compilation scheme, being part of the rules for type-directed compilation and distributive application. The key idea of destinations has been elaborated in Section 6.5.

$$\begin{array}{ll} \text{JavaScript code} & \mathcal{J} ::= \varnothing \mid \mathcal{J}_1; \mathcal{J}_2 \mid \mathsf{code} \\ \text{Type indices} & T ::= \mathbb{Z} \mid \overrightarrow{T} \mid T^{\forall} \mid \{\ell : T\} \mid T_1 \,\&\, T_2 \\ \text{Destinations} & dst ::= \mathbf{nil} \mid y? \mid z \end{array}$$

A.1 Type-Directed Compilation

Similarly to the elaboration in Section 5, the compilation process is type-directed. Besides the typing context Γ, there is also a destination variable dst that guides the code generation. A rule basically reads as: given a typing context Γ and a destination dst, the F_i^+ term e is checked/inferred to have type A and is compiled to variable z in JavaScript code $\mathcal{J}$. That is, after running $\mathcal{J}$, the result is stored in the JavaScript variable z.

Rules J-Int and J-Var have three variants for different destinations, while rules J-App and J-TApp only have one version each but delegate to three variants of application for different destinations, which helps to generate more optimized JavaScript code. Examples illustrating variants of rule J-Var and rule JA-ArrowEquiv (via rule J-App) has been explained in Section 6.5. Rules J-IntOpt and J-IntNil are designed for the optimization of boxing/unboxing (see Section 6.4). Other rules assume that the destination is present and generate code accordingly. Rule J-Nil serves as the bridge from empty destinations to non-empty ones, while rule J-Opt is for optional ones.

$\boxed{\Gamma; dst \vdash e \Leftrightarrow A \rightsquigarrow \mathcal{J} \mid z}$ *(Type-directed compilation)*

J-Nil
$$\frac{\Gamma; z \vdash e \Leftrightarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; \mathbf{nil} \vdash e \Leftrightarrow A \rightsquigarrow \mathsf{code} \mid z}$$

J-Opt
$$\frac{\Gamma; z \vdash e \Leftrightarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; y? \vdash e \Leftrightarrow A \rightsquigarrow \mathsf{code} \mid z}$$

J-Top
$$\frac{}{\Gamma; z \vdash \{\} \Rightarrow \top \rightsquigarrow \varnothing \mid z}$$

J-Int
$$\frac{T = |\mathbb{Z}|}{\Gamma; z \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \mathsf{code} \mid z}$$

J-IntOpt
$$\frac{T = |\mathbb{Z}|}{\Gamma; y? \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \mathsf{code} \mid z}$$

J-IntNil
$$\frac{}{\Gamma; \mathbf{nil} \vdash n \Rightarrow \mathbb{Z} \rightsquigarrow \mathsf{code} \mid z}$$

J-Var
$$\frac{x : A \in \Gamma}{\Gamma; z \vdash x \Rightarrow A \rightsquigarrow \mathsf{code} \mid z}$$

J-VarOpt
$$\frac{x : A \in \Gamma}{\Gamma; y? \vdash x \Rightarrow A \rightsquigarrow \mathsf{code} \mid x}$$

J-VarNil
$$\frac{x : A \in \Gamma}{\Gamma; \mathbf{nil} \vdash x \Rightarrow A \rightsquigarrow \varnothing \mid x}$$

```
/* J-Nil */
var z = {};
J;

/* J-Opt */
var z = y || {};
J;

/* J-Int */
z[T] = n;

/* J-IntOpt */
var z = n;
if (y) y[T] = n;

/* J-IntNil */
var z = n;

/* J-Var */
copy(z, x);

/* J-VarOpt */
if (y) copy(y, x);

/* J-Fix */
var x = z;
J;

/* J-Abs */
z[T] = (x, y) ⇒ {
  J;  return y0;
};

/* J-TAbs */
z[T] = (X, y) ⇒ {
  J;  return y0;
};

/* J-Rcd */
z.__defineGetter__(T, () ⇒ {
  J;
  delete this[T];
  return this[T] = y;
});

/* J-Def */
export var x = {};
J1;  J2;

/* JA-Nil */
var z = {};
J;

/* JA-Opt */
var z = y || {};
J;

/* JA-Arrow */
var y0 = {};
J1;  J2;

/* JA-ArrowEquiv */
x[T](y, z);

/* JA-ArrowOpt */
var z = x[T](y, z0);

/* JA-ArrowNil */
var z = x[T](y);

/* JA-All */
x[T](Ts, z);

/* JA-AllOpt */
var z = x[T](Ts, y);

/* JA-All */
var z = x[T](Ts);

/* JP-RcdEq */
var z = x[T];

/* JS0-Int */
J;
y = y[T];

/* JS0-Var */
J;
if (primitive(X)) y = y[T];

/* JS-Equiv */
copy(y, x);

/* JS-Bot */
y[T] = null;

/* JS-Int */
y[T] = x;

/* JS-IntAnd */
y[T] = x[T];

/* JS-Var */
copy(y, x);

/* JS-Arrow */
y[T2] = (x1, y2) ⇒ {
  var y1 = {};  J1;
  var x2 = x[T1](y1);
  y2 = y2 || {};
  J2;  return y2;
};

/* JS-All */
y[T2] = (X, y0) ⇒ {
  var x0 = x[T1](X);
  y0 = y0 || {};
  J;  return y0;
};

/* JS-Rcd */
y.__defineGetter__(T2, () ⇒ {
  var x0 = x[T1];
  var y0 = {};  J;
  delete this[T];
  return this[T] = y0;
});

/* JS-Split */
var y1 = {}; // if y1 != z
var y2 = {}; // if y2 != z
J1;  J2;  J3;

/* JM-Arrow */
z[T] = (p, y) ⇒ {
  y = y || {};
  var y1 = {}; // if y1 != y
  var y2 = {}; // if y2 != y
  x1[T1](p, y1);
  x2[T2](p, y2);
  J;  return y;
};

/* JM-All */
z[T] = (X, y) ⇒ {
  y = y || {};
  var y1 = {}; // if y1 != y
  var y2 = {}; // if y2 != y
  x1[T1](X, y1);
  x2[T2](X, y2);
  J;  return y;
};

/* JM-Rcd */
z.__defineGetter__(T, () ⇒ {
  var y = {};
  var y1 = {}; // if y1 != y
  var y2 = {}; // if y2 != y
  copy(y1, x1[T1]);
  copy(y2, x2[T2]);
  J;
  delete this[T];
  return this[T] = y;
});
```

Fig. 32. Predefined JavaScript code.

$$\textsc{J-Fix}\quad \frac{\Gamma, x : A; z \vdash e \Leftarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; z \vdash \mathbf{fix}\, x{:}A.\ e \Rightarrow A \rightsquigarrow \mathsf{code} \mid z}$$

$$\textsc{J-TopAbs}\quad \frac{\rceil B \lceil}{\Gamma; z \vdash \lambda x{:}A.\ e{:}B \Rightarrow A \to B \rightsquigarrow \varnothing \mid z}$$

$$\textsc{J-Abs}\quad \frac{T = \overrightarrow{|B|} \qquad \Gamma, x : A; y? \vdash e \Leftarrow B \rightsquigarrow \mathcal{J} \mid y_0}{\Gamma; z \vdash \lambda x{:}A.\ e{:}B \Rightarrow A \to B \rightsquigarrow \mathsf{code} \mid z}$$

$$\textsc{J-App}\quad \frac{\Gamma; \mathbf{nil} \vdash e_1 \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid x \qquad \Gamma; \mathbf{nil} \vdash e_2 \Rightarrow B \rightsquigarrow \mathcal{J}_2 \mid y \qquad \Gamma; dst \vdash x : A \bullet y : B \rightsquigarrow \mathcal{J}_3 \mid z : C}{\Gamma; dst \vdash e_1\, e_2 \Rightarrow C \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2; \mathcal{J}_3 \mid z}$$

$$\textsc{J-TopTAbs}\quad \frac{\rceil B \lceil}{\Gamma; z \vdash \Lambda X{*}A.\ e{:}B \Rightarrow \forall X{*}A.\ B \rightsquigarrow \varnothing \mid z}$$

$$\textsc{J-TAbs}\quad \frac{T = |B|^{\forall} \qquad \Gamma, X * A; y? \vdash e \Leftarrow B \rightsquigarrow \mathcal{J}_2 \mid y_0}{\Gamma; z \vdash \Lambda X{*}A.\ e{:}B \Rightarrow \forall X{*}A.\ B \rightsquigarrow \mathsf{code} \mid z}$$

$$\textsc{J-TApp}\quad \frac{\Gamma; \mathbf{nil} \vdash e \Rightarrow B \rightsquigarrow \mathcal{J}_1 \mid y \qquad \Gamma; dst \vdash y : B \bullet A \rightsquigarrow \mathcal{J}_2 \mid z : C}{\Gamma; dst \vdash e\, A \Rightarrow C \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z}$$

$$\textsc{J-TopRcd}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad \rceil A \lceil}{\Gamma; z \vdash \{\ell = e\} \Rightarrow \{\ell : A\} \rightsquigarrow \varnothing \mid z}$$

$$\textsc{J-Rcd}\quad \frac{T = \{\ell : |A|\} \qquad \Gamma; \mathbf{nil} \vdash e \Rightarrow A \rightsquigarrow \mathcal{J} \mid y}{\Gamma; z \vdash \{\ell = e\} \Rightarrow \{\ell : A\} \rightsquigarrow \mathsf{code} \mid z}$$

$$\textsc{J-Proj}\quad \frac{\Gamma; \mathbf{nil} \vdash e \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid y \qquad y : A \bullet \{\ell\} \rightsquigarrow \mathcal{J}_2 \mid z : B}{\Gamma; z \vdash e.\ell \Rightarrow B \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z}$$

$$\textsc{J-Merge}\quad \frac{\Gamma; z \vdash e_1 \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid z \qquad \Gamma; z \vdash e_2 \Rightarrow B \rightsquigarrow \mathcal{J}_2 \mid z \qquad \Gamma \vdash A * B}{\Gamma; z \vdash e_1 \,,,\, e_2 \Rightarrow A \,\&\, B \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z}$$

$$\textsc{J-Anno}\quad \frac{\Gamma; dst \vdash e \Leftarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; dst \vdash e : A \Rightarrow A \rightsquigarrow \mathcal{J} \mid z}$$

$$\textsc{J-Def}\quad \frac{\Gamma; x \vdash e_1 \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid x \qquad \Gamma, x : A; z \vdash e_2 \Rightarrow B \rightsquigarrow \mathcal{J}_2 \mid z}{\Gamma; z \vdash x = e1; e_2 \Rightarrow B \rightsquigarrow \mathsf{code} \mid z}$$

$$\textsc{J-Sub}\quad \frac{\Gamma; \mathbf{nil} \vdash e \Rightarrow A \rightsquigarrow \mathcal{J}_1 \mid x \qquad x : A <: y : B \rightsquigarrow \mathcal{J}_2}{\Gamma; y \vdash e \Leftarrow B \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid y}$$

$$\textsc{J-SubEquiv}\quad \frac{A \risingdotseq B \qquad \Gamma; dst \vdash e \Rightarrow A \rightsquigarrow \mathcal{J} \mid z}{\Gamma; dst \vdash e \Leftarrow B \rightsquigarrow \mathcal{J} \mid z}$$

A.2 Distributive Application and Projection

We have mentioned that function applications (and record projections) have to be specially handled because of distributive subtyping in F_i^+. To put it simply, we need to additionally consider the cases where functions (and records) have intersection types or top-like types. Similarly to previous rules, destinations also guide the code generation. A rule for function applications basically reads as: given a typing context Γ and a destination *dst*, applying the compiled function in x of type A to the compiled argument p yields variable z of type B in JavaScript code $\mathcal{J}$. Depending on whether the function is λ- or Λ-bound, the argument p can be either a value ($y : C$) or a type (C).

$\boxed{\Gamma; dst \vdash x : A \bullet p \rightsquigarrow \mathcal{J} \mid z : B}$ *(Distributive application)*

$$\text{JA-NIL}\ \frac{\Gamma; z \vdash x : A \bullet p \rightsquigarrow \mathcal{J} \mid z : B}{\Gamma; \mathbf{nil} \vdash x : A \bullet p \rightsquigarrow \mathsf{code} \mid z : B} \qquad \text{JA-OPT}\ \frac{\Gamma; z \vdash x : A \bullet p \rightsquigarrow \mathcal{J} \mid z : B}{\Gamma; y? \vdash x : A \bullet p \rightsquigarrow \mathsf{code} \mid z : B} \qquad \text{JA-TOP}\ \frac{\rceil A \lceil}{\Gamma; z \vdash x : A \bullet p \rightsquigarrow \varnothing \mid z : \top}$$

$$\text{JA-ARROW}\ \frac{T = |\overrightarrow{B}| \quad y : C <: y_0 : A \rightsquigarrow \mathcal{J}_1 \quad \Gamma; dst \vdash x : A \to B \bullet y_0 : A \rightsquigarrow \mathcal{J}_2 \mid z : B}{\Gamma; dst \vdash x : A \to B \bullet y : C \rightsquigarrow \mathsf{code} \mid z : B} \qquad \text{JA-ARROWEQUIV}\ \frac{A \fallingdotseq C \qquad T = |\overrightarrow{B}|}{\Gamma; z \vdash x : A \to B \bullet y : C \rightsquigarrow \mathsf{code} \mid z : B}$$

$$\text{JA-ARROWOPT}\ \frac{A \fallingdotseq C \qquad T = |\overrightarrow{B}|}{\Gamma; z_0? \vdash x : A \to B \bullet y : C \rightsquigarrow \mathsf{code} \mid z : B} \qquad \text{JA-ARROWNIL}\ \frac{A \fallingdotseq C \qquad T = |\overrightarrow{B}|}{\Gamma; \mathbf{nil} \vdash x : A \to B \bullet y : C \rightsquigarrow \mathsf{code} \mid z : B}$$

$$\text{JA-ALL}\ \frac{\Gamma \vdash A * C \quad T = |B|^{\forall} \qquad Ts = \ \mathbf{itoa}\ \mid C \mid}{\Gamma; z \vdash x : \forall X{*}A.\ B \bullet C \rightsquigarrow \mathsf{code} \mid z : B[X \mapsto C]} \qquad \text{JA-ALLOPT}\ \frac{\Gamma \vdash A * C \quad T = |B|^{\forall} \qquad Ts = \ \mathbf{itoa}\ \mid C \mid}{\Gamma; y? \vdash x : \forall X{*}A.\ B \bullet C \rightsquigarrow \mathsf{code} \mid z : B[X \mapsto C]}$$

$$\text{JA-ALLNIL}\ \frac{\Gamma \vdash A * C \quad T = |B|^{\forall} \qquad Ts = \ \mathbf{itoa}\ \mid C \mid}{\Gamma; \mathbf{nil} \vdash x : \forall X{*}A.\ B \bullet C \rightsquigarrow \mathsf{code} \mid z : B[X \mapsto C]} \qquad \text{JA-AND}\ \frac{\Gamma; z \vdash x : A \bullet p \rightsquigarrow \mathcal{J}_1 \mid z : A' \quad \Gamma; z \vdash x : B \bullet p \rightsquigarrow \mathcal{J}_2 \mid z : B'}{\Gamma; z \vdash x : A \& B \bullet p \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z : A' \& B'}$$

As explained in Section 6.4, the rules for record projections are separated to reduce the number of coercions and improve the performance of generated JavaScript code, although they were combined with the rules for function applications in the latest formalization of F_i^+ by Fan et al. [2022]. A rule for record projections basically reads as: projecting the compiled records in x of type A by label ℓ yields variable z of type B in JavaScript code $\mathcal{J}$.

$\boxed{x : A \bullet \{\ell\} \rightsquigarrow \mathcal{J} \mid z : B}$ *(Distributive projection)*

$$\text{JP-TOP}\ \frac{\rceil A \lceil}{x : A \bullet \{\ell\} \rightsquigarrow \varnothing \mid z : \top} \qquad \text{JP-RCDEQ}\ \frac{T = \{\ell : |A|\}}{x : \{\ell : A\} \bullet \{\ell\} \rightsquigarrow \mathsf{code} \mid z : A} \qquad \text{JP-RCDNEQ}\ \frac{\ell_1 \neq \ell_2 \qquad T = \{\ell : |A|\}}{x : \{\ell_1 : A\} \bullet \{\ell_2\} \rightsquigarrow \varnothing \mid z : \top}$$

$$\text{JP-AND}\ \frac{x : A \bullet \{\ell\} \rightsquigarrow \mathcal{J}_1 \mid z : A' \quad x : B \bullet \{\ell\} \rightsquigarrow \mathcal{J}_2 \mid z : B'}{x : A \& B \bullet \{\ell\} \rightsquigarrow \mathcal{J}_1; \mathcal{J}_2 \mid z : A' \& B'}$$

A.3 Coercive Subtyping

In rule J-SUB, we check an expression of type A against its supertype B. Since the two types may correspond to compiled objects of different shapes, a coercion has to be inserted for each subtyping check. Such a form of subtyping is called *coercive* subtyping [Luo et al. 2013], in contrast to *inclusive* subtyping. A rule for coercive subtyping basically reads as: to upcast a compiled object x of type A to a compiled object y of type B, we need to insert a coercion in JavaScript code $\mathcal{J}$. The umbrella rule has three variants because of the optimization of boxing/unboxing (see Section 6.4).

$\boxed{x : A <: y : B \rightsquigarrow \mathcal{J}}$ *(Coercive subtyping)*

$$\textsc{JS0-Sub}\quad \frac{x : A <:^{+} y : B \rightsquigarrow \mathcal{J}}{x : A <: y : B \rightsquigarrow \mathcal{J}}$$

$$\textsc{JS0-Int}\quad \frac{\begin{array}{c} T = |\mathbb{Z}| \\ x : A <:^{+} y : \mathbb{Z} \rightsquigarrow \mathcal{J} \end{array}}{x : A <: y : \mathbb{Z} \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS0-Var}\quad \frac{\begin{array}{c} T = |X| \\ x : A <:^{+} y : X \rightsquigarrow \mathcal{J} \end{array}}{x : A <: y : X \rightsquigarrow \mathsf{code}}$$

As explained in Section 6.4, we add an extra flag to help optimize coercions for subtyping between equivalent types: $<:^{+}$ indicates that the optimization rule JS-Equiv can apply, while $<:^{-}$ not.

$\boxed{x : A <:^{\pm} y : B \rightsquigarrow \mathcal{J}}$ *(Coercive subtyping)*

$$\textsc{JS-Equiv}\quad \frac{A \fallingdotseq B}{x : A <:^{+} y : B \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS-Top}\quad \frac{\rceil B \lceil}{x : A <:^{\pm} y : B \rightsquigarrow \varnothing}$$

$$\textsc{JS-Bot}\quad \frac{T = |A|}{x : \bot <:^{\pm} y : A \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS-Int}\quad \frac{}{x : \mathbb{Z} <:^{+} y : \mathbb{Z} \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS-IntAnd}\quad \frac{T = |\mathbb{Z}|}{x : \mathbb{Z} <:^{-} y : \mathbb{Z} \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS-Var}\quad \frac{}{x : X <:^{\pm} y : X \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS-Arrow}\quad \frac{\begin{array}{c} T_1 = |\overrightarrow{A_2}| \\ T_2 = |\overrightarrow{B_2}| \qquad Ts = \mathbf{itoa}\ |\ A_1\ | \\ x_1 : B_1 <: y_1 : A_1 \rightsquigarrow \mathcal{J}_1 \\ x_2 : A_2 <: y_2 : B_2 \rightsquigarrow \mathcal{J}_2 \end{array}}{x : A_1 \to A_2 <:^{\pm} y : B_1 \to B_2 \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS-All}\quad \frac{\begin{array}{c} T_1 = |A_2|^{\forall} \\ T_2 = |B_2|^{\forall} \qquad B_1 <: A_1 \\ x_0 : A_2 <: y_0 : B_2 \rightsquigarrow \mathcal{J} \end{array}}{x : \forall X{*}A_1.\, A_2 <:^{\pm} y : \forall X{*}B_1.\, B_2 \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS-Rcd}\quad \frac{\begin{array}{c} T_1 = \{\ell : |A|\} \\ T_2 = \{\ell : |B|\} \\ x_0 : A <: y_0 : B \rightsquigarrow \mathcal{J} \end{array}}{x : \{\ell : A\} <:^{\pm} y : \{\ell : B\} \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS-Split}\quad \frac{\begin{array}{c} B_1 \lhd B \rhd B_2 \\ y_1 : B_1 \rhd z : B \lhd y_2 : B_2 \rightsquigarrow \mathcal{J}_3 \\ x : A <:^{\pm} y_1 : B_1 \rightsquigarrow \mathcal{J}_1 \\ x : A <:^{\pm} y_2 : B_2 \rightsquigarrow \mathcal{J}_2 \end{array}}{x : A <:^{\pm} z : B \rightsquigarrow \mathsf{code}}$$

$$\textsc{JS-AndL}\quad \frac{x : A <:^{-} y : C \rightsquigarrow \mathcal{J}}{x : A \,\&\, B <:^{\pm} y : C \rightsquigarrow \mathcal{J}}$$

$$\textsc{JS-AndR}\quad \frac{x : B <:^{-} y : C \rightsquigarrow \mathcal{J}}{x : A \,\&\, B <:^{\pm} y : C \rightsquigarrow \mathcal{J}}$$

There are some auxiliary rules called *coercive merging* for rule JS-Split. These rules mean that if the splitting relation $A \lhd C \rhd B$ holds, we can merge the compiled objects x of type A and y of type B back into a single object z of type C in JavaScript code $\mathcal{J}$. Such merging is necessary after splitting the supertype distributively. For example, consider the following derivation of subtyping:

$$\frac{\begin{array}{c} \top \to \mathbf{Int} \lhd \top \to \mathbf{Int} \,\&\, \mathbf{Bool} \rhd \top \to \mathbf{Bool} \\ \top \to \mathbf{Int} \,\&\, \mathbf{String} \,\&\, \mathbf{Bool} <: \top \to \mathbf{Int} \qquad \top \to \mathbf{Int} \,\&\, \mathbf{String} \,\&\, \mathbf{Bool} <: \top \to \mathbf{Bool} \end{array}}{\top \to \mathbf{Int} \,\&\, \mathbf{String} \,\&\, \mathbf{Bool} <: \top \to \mathbf{Int} \,\&\, \mathbf{Bool}}$$

After splitting, the compiled object would have two fields with labels `"fun_int"` and `"fun_bool"`, but we expect only one field with label `"fun_(int&bool)"`. Rule JM-Arrow handles this case and merge the two fields back into one.

The notation may be misleading, but note that here only the variable name z is given (i.e. input) while variable names x and y are generated by the rules (i.e. output). This is because rule JM-And reuses the variable name z to also serve as x and y, which makes the caller perform more efficient in-place updates. Not to make it more confusing but we have to emphasize that the discussion is only about the variable *name* rather than the contents of the variable. Having a closer look at rule JS-Split will help to better understand our design. Below the judgment of coercive merging, the generated variable names (y_1 and y_2 in the case) are used to generate the coercions (in $\mathcal{J}_1$ and $\mathcal{J}_2$). The coercions are actually executed before the coercive merging (in $\mathcal{J}_3$) in generated JavaScript. To avoid y_1 and y_2 from being initialized more than once, some extra checks are performed when generating JavaScript code for rules JS-Split, JM-Arrow, JM-All, and JM-Rcd.

$\boxed{x : A \ \triangleright\ z : C\ \triangleleft\ y : B \rightsquigarrow \mathcal{J}}$ *(Coercive merging)*

JM-And

$$\frac{}{z : A \ \triangleright\ z : A \,\&\, B\ \triangleleft\ z : B \rightsquigarrow \varnothing}$$

JM-Arrow

$$\frac{T = \overrightarrow{|B|} \qquad T_1 = \overrightarrow{|B_1|} \qquad T_2 = \overrightarrow{|B_2|} \qquad y_1 : B_1 \ \triangleright\ y : B\ \triangleleft\ y_2 : B_2 \rightsquigarrow \mathcal{J}}{x_1 : A \to B_1 \ \triangleright\ z : A \to B\ \triangleleft\ x_2 : A \to B_2 \rightsquigarrow \mathsf{code}}$$

JM-All

$$\frac{T = |B|^{\forall} \qquad T_1 = |B_1|^{\forall} \qquad T_2 = |B_2|^{\forall} \qquad y_1 : B_1 \ \triangleright\ y : B\ \triangleleft\ y_2 : B_2 \rightsquigarrow \mathcal{J}}{x_1 : \forall X{*}A.\ B_1 \ \triangleright\ z : \forall X{*}A.\ B\ \triangleleft\ x_2 : \forall X{*}A.\ B_2 \rightsquigarrow \mathsf{code}}$$

JM-Rcd

$$\frac{T = \{\ell : |A|\} \qquad T_1 = \{\ell : |A_1|\} \qquad T_2 = \{\ell : |A_2|\} \qquad y_1 : A_1 \ \triangleright\ y : A\ \triangleleft\ y_2 : A_2 \rightsquigarrow \mathcal{J}}{x_1 : \{\ell : A_1\} \ \triangleright\ z : \{\ell : A\}\ \triangleleft\ x_2 : \{\ell : A_2\} \rightsquigarrow \mathsf{code}}$$

Acknowledgments

We thank Utkarsh Dhandhania for his contribution to the CP compiler. We are grateful to Jonathan Aldrich and the anonymous reviewers for their constructive feedback, which helped us improve the paper. This work is supported by Hong Kong Research Grant Council under project number 17209821.

References

João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. 2017. Disjoint Polymorphism. In *ESOP*. doi:10.1007/978-3-662-54434-1_1

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013a. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer. doi:10.1007/978-3-642-37521-7

Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *J. Object Technol.* 8, 5 (2009). doi:10.5381/jot.2009.8.5.c5

Sven Apel, Christian Kästner, and Christian Lengauer. 2013b. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Eng.* 39, 1 (2013). doi:10.1109/TSE.2011.120

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. 2005. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE*. doi:10.1007/11561347_10

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. 2006. An overview of CaesarJ. *Trans. Aspect Oriented Softw. Dev.* 1 (2006). doi:10.1007/11687061_5

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symb. Log.* 48, 4 (1983). doi:10.2307/2273659

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. 1996. A Monotonic Superclass Linearization for Dylan. In *OOPSLA*. doi:10.1145/236337.236343

Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.* 30, 6 (2004). doi:10.1109/TSE.2004.23

Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. 2013. Compositional Type Checking of Delta-Oriented Software Product Lines. *Acta Informatica* 50, 2 (2013). doi:10.1007/s00236-012-0173-z

Xuan Bi and Bruno C. d. S. Oliveira. 2018. Typed First-Class Traits. In *ECOOP*. doi:10.4230/LIPIcs.ECOOP.2018.9

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The Essence of Nested Composition. In *ECOOP*. doi:10.4230/LIPIcs.ECOOP.2018.22

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *ESOP*. doi:10.1007/978-3-030-17184-1_14

Dariusz Biernacki and Piotr Polesiuk. 2015. Logical Relations for Coherence of Effect Subtyping. In *TLCA*. doi:10.4230/LIPIcs.TLCA.2015.107

Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. In *OOPSLA/ECOOP*. doi:10.1145/97945.97982

Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *ECOOP*. doi:10.1007/978-3-642-14107-2_20

Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. 1995. On Binary Methods. *Theory Pract. Object Sys.* 1, 3 (1995). doi:10.1002/j.1096-9942.1995.tb00019.x

Kim B. Bruce. 2002. *Foundations of Object-Oriented Languages: Types and Semantics.* MIT press.

Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. 1999. Comparing Object Encodings. *Inf. Comput.* 155, 1-2 (1999). doi:10.1006/inco.1999.2829

Luca Cardelli and John C. Mitchell. 1991. Operations on Records. *Math. Struct. Comput. Sci.* 1, 1 (1991). doi:10.1017/S0960129500000049

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (1985). doi:10.1145/6041.6042

Craig Chambers. 1992. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* Ph. D. Dissertation. Stanford University.

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: A Simple Virtual Class Calculus. In *AOSD*. doi:10.1145/1218563.1218578

William Cook and Jens Palsberg. 1989. A Denotational Semantics of Inheritance and Its Correctness. In *OOPSLA*. doi:10.1145/74878.74922

William R. Cook, Walter L. Hill, and Peter S. Canning. 1990. Inheritance Is Not Subtyping. In *POPL*. doi:10.1145/96709.96721

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991). doi:10.1145/115372.115320

Ferruccio Damiani, Luca Padovani, Ina Schaefer, and Christoph Seidl. 2018. A Core Calculus for Dynamic Delta-Oriented Programming. *Acta Informatica* 55, 4 (2018). doi:10.1007/s00236-017-0293-6

N. G. de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972). doi:10.1016/1385-7258(72)90034-0

Karel Driesen, Urs Hölzle, and Jan Vitek. 1995. Message Dispatch on Pipelined Processors. In *ECOOP*. doi:10.1007/3-540-49538-X_13

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A Mechanism for Fine-Grained Reuse. *ACM Trans. Program. Lang. Syst.* 28, 2 (2006). doi:10.1145/1119479.1119483

Jana Dunfield. 2014. Elaborating Intersection and Union Types. *J. Funct. Program.* 24, 2–3 (2014). doi:10.1017/S0956796813000270

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

ECMA. 1999. ECMA-262 3rd Edition, ECMAScript Language Specification. https://ecma-international.org/wp-content/uploads/ECMA-262_3rd_edition_december_1999.pdf

ECMA. 2015. ECMA-262 6th Edition, ECMAScript 2015 Language Specification. https://262.ecma-international.org/6.0/

Erik Ernst. 2000. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* Ph. D. Dissertation. Aarhus University. doi:10.7146/dpb.v29i549.7654

Erik Ernst. 2001. Family Polymorphism. In *ECOOP*. doi:10.1007/3-540-45337-7_17

Erik Ernst. 2002. Safe Dynamic Multiple Inheritance. *Nordic J. Comput.* 9, 3 (2002).

Erik Ernst. 2004. The Expression Problem, Scandinavian Style. In *MASPEGHI@ECOOP*. doi:10.1007/978-3-540-30554-5_11

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A Virtual Class Calculus. In *POPL*. doi:10.1145/1111037.1111062

Andong Fan, Xuejing Huang, Han Xu, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2022. Direct Foundations for Compositional Programming. In *ECOOP*. doi:10.4230/LIPIcs.ECOOP.2022.18

Benedict R. Gaster and Mark P. Jones. 1996. *A Polymorphic Type System for Extensible Records and Variants*. Technical Report NOTTCS-TR-96-3. University of Nottingham.

Neal Glew. 1999. Type Dispatch for Named Hierarchical Types. In *ICFP*. doi:10.1145/317636.317797

Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernando Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight Go. In *OOPSLA*. doi:10.1145/3428217

Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (2008). doi:10.1109/MC.2008.123

Robert Harper and Benjamin Pierce. 1991. A Record Calculus Based on Symmetric Concatenation. In *POPL*. doi:10.1145/99583.99603

Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. Taming the Merge Operator. *J. Funct. Program.* 31 (2021). doi:10.1017/S0956796821000186

Paul Hudak. 1998. Modular Domain Specific Languages and Tools. In *ICSR*. doi:10.1109/ICSR.1998.685738

Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP*. doi:10.1007/BFb0057013

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001). doi:10.1145/503502.503505

Mark P. Jones. 1994. A Theory of Qualified Types. *Sci. Comput. Program.* 22, 3 (1994). doi:10.1016/0167-6423(94)00005-0

Anastasiya Kravchuk-Kirilyuk, Gary Feng, Jonas Iskander, Yizhou Zhang, and Nada Amin. 2024. Persimmon: Nested Family Polymorphism with Extensible Variant Types. In *OOPSLA*. doi:10.1145/3649836

Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The Road to Feature Modularity?. In *FOSD@SPLC*. doi:10.1145/2019136.2019142

Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A Theory of Tagged Objects. In *ECOOP*. doi:10.4230/LIPIcs.ECOOP.2015.174

Daan Leijen. 2004. *First-Class Labels for Extensible Rows*. Technical Report UU-CS-2004-051. Utrecht University.

Daan Leijen. 2005. Extensible Records with Scoped Labels. In *TFP*.

Roberto E. Lopez-Herrejon, Don Batory, and William Cook. 2005. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*. doi:10.1007/11531142_8

Zhaohui Luo, Sergei Soloviev, and Tao Xue. 2013. Coercive Subtyping: Theory and Implementation. *Inf. Comput.* 223 (2013). doi:10.1016/j.ic.2012.10.020

Ole Lehrmann Madsen and Birger Møller-Pedersen. 1989. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *OOPSLA*. doi:10.1145/74877.74919

Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. 1993. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley.

Koar Marntirosian, Tom Schrijvers, Bruno C. d. S. Oliveira, and Georgios Karachalias. 2020. Resolution as Intersection Subtyping via Modus Ponens. In *OOPSLA*. doi:10.1145/3428274

Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. 2001. Jiazzi: New-Age Components for Old-Fashioned Java. In *OOPSLA*. doi:10.1145/504282.504298

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

Leonid Mikhajlov and Emil Sekerinski. 1998. A Study of The Fragile Base Class Problem. In *ECOOP*. doi:10.1007/BFb0054099

Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. 2004. Scalable Extensibility via Nested Inheritance. In *OOPSLA*. doi:10.1145/1028976.1028986

Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. 2006. J&: Nested Intersection for Scalable Software Composition. In *OOPSLA*. doi:10.1145/1167473.1167476

Martin Odersky and Matthias Zenger. 2005. Scalable Component Abstractions. In *OOPSLA*. doi:10.1145/1094811.1094815

Atsushi Ohori. 1995. A Polymorphic Record Calculus and Its Compilation. *ACM Trans. Program. Lang. Syst.* 17, 6 (1995). doi:10.1145/218570.218572

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *ECOOP*. doi:10.1007/978-3-642-31057-7_2

Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint Intersection Types. In *ICFP*. doi:10.1145/2951913.2951945

Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. 2013. Feature-Oriented Programming with Object Algebras. In *ECOOP*. doi:10.1007/978-3-642-39038-8_2

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000). doi:10.1145/345099.345100

Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*. doi:10.1007/BFb0053389

Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2014. From Object Algebras to Attribute Grammars. In *OOPSLA*. doi:10.1145/2660193.2660237

John C. Reynolds. 1997. Design of the Programming Language Forsythe. In *Algol-like Languages*. Vol. 1. Chapter 8. doi:10.1007/978-1-4612-4118-8_9

Chieri Saito, Atsushi Igarashi, and Mirko Viroli. 2008. Lightweight Family Polymorphism. *J. Funct. Program.* 18, 3 (2008). doi:10.1017/S0956796807006405

Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *SPLC*. doi:10.1007/978-3-642-15579-6_6

Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-Passing Style for Efficient Memory Management. In *FHPC@ICFP*. doi:10.1145/3122948.3122949

Mark Shields and Erik Meijer. 2001. Type-Indexed Rows. In *POPL*. doi:10.1145/360204.360230

T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. 2013. Contracts for First-Class Classes. *ACM Trans. Program. Lang. Syst.* 35, 3 (2013). doi:10.1145/2518189

Yaozhu Sun. 2025. *Compositional Programming in Action*. Ph. D. Dissertation. The University of Hong Kong. https://hub.hku.hk/handle/10722/358307

Yaozhu Sun, Utkarsh Dhandhania, and Bruno C. d. S. Oliveira. 2022. Compositional Embeddings of Domain-Specific Languages. In *OOPSLA*. doi:10.1145/3563294

W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967). doi:10.2307/2271658

Antero Taivalsaari. 1996. On the Notion of Inheritance. *ACM Comput. Surv.* 28, 3 (1996). doi:10.1145/243439.243441

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. In *OOPSLA*. doi:10.1145/2384616.2384674

David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *OOPSLA*. doi:10.1145/38765.38828

Philip Wadler. 1998. The Expression Problem. Posted on the Java Genericity mailing list. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

Mitchell Wand. 1991. Type Inference for Record Concatenation and Multiple Inheritance. *Inf. Comput.* 93, 1 (1991). doi:10.1016/0890-5401(91)90050-C

Ningning Xie, Bruno C. d. S. Oliveira, Xuan Bi, and Tom Schrijvers. 2020. Row and Bounded Polymorphism via Disjoint Polymorphism. In *ECOOP*. doi:10.4230/LIPIcs.ECOOP.2020.27

Han Xu, Xuejing Huang, and Bruno C. d. S. Oliveira. 2023. Making a Type Difference: Subtraction on Intersection Types as Generalized Record Operations. In *POPL*. doi:10.1145/3571224

Matthias Zenger and Martin Odersky. 2005. Independently Extensible Solutions to the Expression Problem. In *FOOL@POPL*.

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3 (2021). doi:10.1145/3460228

Yizhou Zhang and Andrew C. Myers. 2017. Familia: Unifying Interfaces, Type Classes, and Family Polymorphism. In *OOPSLA*. doi:10.1145/3133894