Deriving Dependently-Typed OOP from First Principles

DAVID BINDER, University of Tübingen, Germany
INGO SKUPIN, University of Tübingen, Germany
TIM SÜBERKRÜB, Aleph Alpha Research at IPAI, Germany
KLAUS OSTERMANN, University of Tübingen, Germany

The *expression problem* describes how most types can easily be extended with new ways to *produce* the type or new ways to *consume* the type, but not both. When abstract syntax trees are defined as an algebraic data type, for example, they can easily be extended with new consumers, such as *print* or *eval*, but adding a new constructor requires the modification of all existing pattern matches. The expression problem is one way to elucidate the difference between functional or data-oriented programs (easily extendable by new consumers) and object-oriented programs (easily extendable by new producers). This difference between programs which are extensible by new producers or new consumers also exists for dependently typed programming, but with one core difference: Dependently-typed programming almost exclusively follows the functional programming model and not the object-oriented model, which leaves an interesting space in the programming language landscape unexplored. In this paper, we explore the field of dependently-typed object-oriented programming by *deriving it from first principles* using the principle of duality. That is, we do not extend an existing object-oriented formalism with dependent types in an ad-hoc fashion, but instead start from a familiar data-oriented language and derive its dual fragment by the systematic use of defunctionalization and refunctionalization. Our central contribution is a dependently typed calculus which contains two dual language fragments. We provide type- and semantics-preserving transformations between these two language fragments: defunctionalization and refunctionalization. We have implemented this language and these transformations and use this implementation to explain the various ways in which constructions in dependently typed programming can be explained as special instances of the general phenomenon of duality.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Software verification**; *Data types and structures*; *Classes and objects.*

Additional Key Words and Phrases: Dependent Types, Expression Problem, Defunctionalization, Codata Types

ACM Reference Format:
David Binder, Ingo Skupin, Tim Süberkrüb, and Klaus Ostermann. 2024. Deriving Dependently-Typed OOP from First Principles. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 129 (April 2024), 27 pages. https://doi.org/10.1145/3649846

1 INTRODUCTION

There are many programming paradigms, but dependently typed programming languages almost exclusively follow the functional programming model. In this paper, we show why dependently-typed programming languages should also include object-oriented principles, and how this can

Authors' addresses: David Binder, Department of Computer Science, University of Tübingen, Sand 14, Tübingen, 72076, Germany, david.binder@uni-tuebingen.de; Ingo Skupin, Department of Computer Science, University of Tübingen, Sand 14, Tübingen, 72076, Germany, skupin@informatik.uni-tuebingen.de; Tim Süberkrüb, Aleph Alpha Research at IPAI, Grenzhöfer Weg 36, Heidelberg, 69123, Germany, tim.sueberkrueb@aleph-alpha-ip.ai; Klaus Ostermann, Department of Computer Science, University of Tübingen, Sand 14, Tübingen, 72076, Germany, klaus.ostermann@uni-tuebingen.de.

ACM 2475-1421/2024/4-ART129
https://doi.org/10.1145/3649846

be done. One of the main reasons why object-oriented features should be included is a consequence of how the complexity of the domain is modeled in the functional and object-oriented paradigm. Functional programmers structure the domain using data types defined by their constructors, whereas object-oriented programmers structure the domain using classes and interfaces defined by methods. This choice has important implications for the extensibility properties of large programs, which are only more accentuated for dependently typed programs.

Why do most dependently typed languages follow the functional style? One of the main reasons is that dependent type theories, on which a lot of them are based, are best studied for functional programming languages. Our challenge, then, is to develop a dependently-typed object-oriented calculus that can serve as the foundation for object-oriented dependently-typed programming languages. Instead of specifying this calculus in an ad-hoc fashion, we want to use de- and refunctionalization as systematic tools to *derive* an object-oriented language fragment from its functional counterpart. We want to show how object-oriented programming is *dual* to functional programming, that this duality extends from non-dependent programming languages to dependently typed programming languages, and that we can use this duality to derive our calculus.

1.1 Data and Codata: The Essence of Functional and Object-Oriented Programming

How can functional programming (FP) and object-oriented programming (OOP) be dual, if there is no precise definition of these two paradigms? We have to define what we mean by functional and object-oriented programming if we want to get a precise research question. For the purposes of this paper, and other reasonable definitions notwithstanding, we focus on the differences in program decomposition between the two paradigms.[1] For us, the essence of functional programming is programming with algebraic data types and pattern matching, whereas the essence of object-oriented programming is programming against interfaces, which correspond to the type-theoretic concept of codata and copattern matching. This definition is not novel but follows similar observations by Cook [1990, 2009] and Downen et al. [2019]. A potentially confusing but important aspect of this definition is that first-class functions are in the object-oriented space, since they are a particular form of codata (and λ is a particular form of copattern matching). In the rest of this subsection, we elaborate on this definition.

Let us verify first that this definition captures the essence of FP. An essential part of the programming experience in statically typed functional languages like OCaml, Scala, Haskell or SML, but also proof assistants like Coq, Agda, Idris and Lean, is modeling the domain with algebraic data types. Algebraic data types consist of product types like structs and records, sum types and enums, and recursive types like lists, which together form the essential vocabulary with which programmers in those languages express themselves. The dependently typed languages in this list extend this vocabulary by allowing data types to be *indexed*; the vector type, for example, is indexed over the number of its elements.

That OOP can be identified with codata types is less obvious, so we will introduce them with a bit more detail. Data types and codata types differ in how they are defined: Whereas a data type is defined by its constructors, i.e. all the ways in which terms of that type can be constructed, a codata type is defined by all the ways it can be observed. One type which is defined by its two canonical observations is the type of infinite streams. We can either observe the head of a stream, yielding one element, or we can observe the tail, yielding a new stream. Equivalently, we can say that every stream has to implement the stream interface which requires a head and a tail method.

[1] Such a definition necessarily reduces the differences between the two paradigms to only one aspect, but this reduction is hopefully also illuminating. Focusing on another difference, and, for example, analyzing how subtyping can influence the design of dependently typed programming language would be another interesting research question.

Instead of this object-oriented terminology, we use the type-theoretic jargon and the following syntax for defining the type of streams:

```
codata Stream(a: Type) {
  Stream(a).head(a: Type): a,
  Stream(a).tail(a: Type): Stream(a) }
```

```
codef Ones: Stream(Nat) {
  head(_) => S(Z),
  tail(_) => Ones }
```

The right-hand side shows how to construct a stream by implementing the stream interface, i.e. by saying how it will behave on the head and the tail observation. This particular stream models an infinite sequence of ones. The syntactic construct we use here is called copattern matching [Abel et al. 2013] and is the precise dual of pattern matching.

We mentioned that according to our definition of FP and OOP, first-class functions counter-intuitively belong to the object-oriented space, so let us substantiate that claim. Programmers in functional programming languages can define many types, but they usually cannot define the function type. Functions can be defined, however, using codata types: A function is just an object which implements an interface with one apply method. For example, functions from natural numbers to Booleans, and the constant function which always returns true are defined in the following way:

```
codata Fun { ap(x: Nat): Bool }
```

```
codef ConstTrue: Fun { ap(_) => True }
```

The research question that motivated this paper is this: If functional programming can be and has been extended to dependent functional programming, can object-oriented programming be similarly extended? Codata has been introduced to many proof assistants before, but for an entirely different purpose. The purpose was to model certain infinite structures and coinductive objects, not to program in an object-oriented style. In this paper, we are interested in this second aspect, and we are (to the best of our knowledge) the first ones to discuss this question in detail. To approach this question in a principled way, we need an additional technical tool, defunctionalization and refunctionalization, which we introduce in the next section.

1.2 De- and Refunctionalization: A Tool for Systematic Language Design

Now that we have introduced two alternative programming paradigms, let us look at how one paradigm can express programs in the other paradigm. One way in which object-oriented programmers have often represented the functional style is with the visitor pattern [Gamma et al. 1995]. Later, Downen et al. [2019] showed how the visitor pattern can be used as a compilation technique for data and codata types; using the visitor pattern, they can compile functional programs to object-oriented programs, and using a related tabulation technique they can compile object-oriented programs to functional ones. In this paper, we use an alternative technique: defunctionalization and refunctionalization.

Defunctionalization [Danvy and Nielsen 2001; Reynolds 1972] is a whole-program transformation which eliminates higher-order functions by replacing lambda abstractions by constructors of a data type, together with a top-level `apply` function. Refunctionalization [Danvy and Millikin 2009] is its partial inverse, and re-introduces higher-order functions by replacing occurrences of the constructors by lambda abstractions. We already observed in the previous section that the function type is just one instance of a codata type. Based on this observation, Rendel et al. [2015] showed that defunctionalization and refunctionalization can be generalized to arbitrary data and codata types, which makes these transformations both more powerful and more symmetric since refunctionalization is now a full inverse instead of a partial one.

Let us look at an example of how these generalized defunctionalization and refunctionalization transformations work. In Figure 1a we have defined Booleans as a data type with two constructors, and negation by pattern matching on `True` and `False`. For negation we use syntax familiar to object-oriented programmers: negating a boolean b can be written as b.`neg`. Refunctionalizing

this program results in the program in Figure 1b. In this representation, negation is the single observation of a codata type, and `True` and `False` are defined as objects implementing this interface.

```
data Bool { True, False }
def Bool.neg: Bool {
  True => False,
  False => True }
```

(a) Functional programming style.

```
codata Bool { neg: Bool }
codef True: Bool { neg => False }
codef False: Bool { neg => True }
```

(b) Object-oriented style.

Fig. 1. Two representations of the same program.

One way to visualize how defunctionalization and refunctionalization work is to think of each type as a matrix. The two programs of Figure 1, for example, can be represented by the following matrix:

Bool	*True*	*False*
neg	False	True

The rows of the matrix enumerate all the ways elements of the type can be consumed, whereas the columns enumerate the ways in which elements of the type can be constructed. The cells of the matrix specify the result of an interaction between one of each. Data types and codata types are then just two different linear presentations of this type-matrix, and defunctionalization and refunctionalization transpose the linearization.

In this paper, we use defunctionalization and refunctionalization not as a compilation technique, but as a *tool for systematic language design*. These transformations are only total in a language where the data and codata fragments of the language are equally expressive. We can therefore use them to systematically *derive* the codata fragment of an object-oriented dependently-typed language by starting from a familiar design for dependent data types and pattern matching, and refunctionalizing programs in that language.

1.3 A Minimal Dependently-Typed Example

Let us now extend the example from the previous section by a simple proof that negation is an involution, i.e. that applying negation twice is the identity. We look at this example first from the familiar point of view of functional programming, and then from the more unfamiliar point of view of dependently-typed object-oriented programming. These two dual presentations are not artificially constructed but inter-derived using de- and refunctionalization introduced in the previous section. In the accompanying implementation that we provide, each version can be automatically transformed into the other presentation at the click of a button.

In the functional decomposition, shown in Figure 2a, we use the Martin-Löf equality type $\texttt{Eq}(a : \text{Type}, x\ y : a)$ to express propositional equality. The way we defined the proposition that negating a boolean twice is the identity function is interesting. Instead of a dependent function, it is formulated more directly as an elimination on a named boolean `self` which yields a proof that `self` is equal to `self` twice-negated, i.e. `self.neg.neg`. The proof pattern matches on `True` and `False` and returns the `Refl` constructor in each branch.

```
data Eq(a: Type, x y: a) {
  Refl(a: Type, x: a): Eq(a, x, x) }
data Bool { True, False }
def Bool.neg: Bool {
  True => False,
  False => True }
def (self: Bool).neg_inverse
  : Eq(Bool, self, self.neg.neg) {
  True => Refl(Bool, True),
  False => Refl(Bool, False) }
```

(a) Functional programming style.

```
data Eq(a: Type, x y: a) {
  Refl(a: Type, x: a): Eq(a, x, x) }
codata Bool {
  neg: Bool,
  (self: Bool).neg_inverse
    : Eq(Bool, self, self.neg.neg) }
codef True: Bool {
  neg => False,
  neg_inverse => Refl(Bool, True) }
codef False: Bool {
  neg => True,
  neg_inverse => Refl(Bool, False) }
```

(b) Object-oriented style.

Fig. 2. Extending Figure 1 with proofs.

In the object-oriented decomposition, shown in Figure 2b, we have kept the definition of the Martin-Löf equality type. The definition of Booleans, on the other hand, has changed dramatically. Booleans are now defined via the two observations that we defined in the original program: negation and the proof that negating a boolean twice is the identity. Instead of two canonical constructors `True` and `False` we now have two mutually recursive top-level definitions of `True` and `False`. This means that we are now free to add new Booleans without changing the definition of the type `Bool`, thus we could just add another object and implement the negation operation together with a proof of its correctness, i.e. a proof that applying it twice yields the original element.

Defining objects by interfaces that they have to implement is of course familiar. However, also including proofs of correctness in those interfaces, and looking at familiar types like Booleans in this flipped representation is novel. In this article, we invite you to follow us on an exploration of the duality of these two programming styles and to discover both the expressive power we get and the sometimes subtle problems we encounter and the restrictions we have to impose.

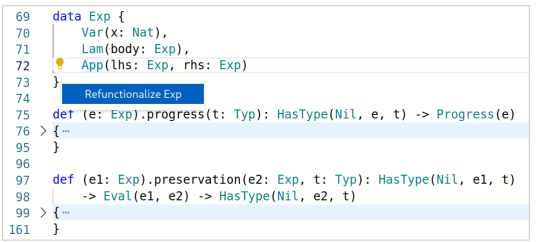

Fig. 3. Screenshot of the online IDE available at polarity-lang.github.io/oopsla24.

1.4 Overview

The remainder of this article is structured as follows:

- Building on our minimal example, we present dependently typed object-oriented programming in Section 2. Our language consists of two fragments, a functional/data-oriented fragment and an object-oriented/codata fragment, and the specification of these two fragments is dictated by the requirement that defunctionalization and refunctionalization are total and semantic-preserving transformations.
- In Section 3 we evaluate the expressive power and the extensibility properties of our system using a case study of a dependently typed web server. Furthermore, in the artifact Binder et al. [2024a] which accompanies this article, we evaluate our design and implementation by a formalization of type preservation for a simple expression language respectively full type soundness of the simply typed lambda calculus. Since we have an available online implementation, the reader can choose to flip any of the involved types from the data to the codata representation, and vice-versa, and observe the resulting program.
- In Section 4 we discuss the constraints on the design of the type system that we had to observe because we want our system to be closed under de- and refunctionalization. For each such constraint, we discuss both the problem and the solution that we have chosen for our formalization.
- In Sections 5 and 6 we present all the formal details. We specify a declarative formalization in the style of Martin-Löf in Section 5 and the details of the defunctionalization and refunctionalization algorithms in Section 6.
- We discuss future work in Section 7, related work in Section 8 and conclude in Section 9.

We have implemented the language, and the defunctionalization and refunctionalization algorithms presented in this paper. We make an IDE available in the browser (cf. Figure 3) which supports the defunctionalization and refunctionalization transformations as code actions using the language server protocol (LSP).

2 DEPENDENTLY-TYPED OBJECT-ORIENTED PROGRAMMING

Typical programs written in object-oriented and functional languages have many differences. We will now look at how these differences appear when we consider programs written in the object-oriented style.

2.1 Method Call Syntax and Self Parameters

Object-oriented programmers are familiar with the *method call syntax* $o.\mathtt{f}(e)$ to invoke a method `f` taking an argument e on an object o. This is sometimes presented under the name "uniform function call syntax" as an alternative notation for the call $\mathtt{f}(o, e)$, where the first argument of the method `f` is called the *self parameter*. We take this simple syntactic observation, see how we have to modify it to the dependently typed setting, and how this influenced our design of codata types.

```
import Agda.Builtin.Equality
open Agda.Builtin.Equality

data Bool : Set where
  true : Bool
  false : Bool

neg : Bool -> Bool
neg true = false
neg false = true

neg_inverse
  : (x : Bool) -> x ≡ neg (neg x)
neg_inverse true = refl
neg_inverse false = refl
```

Fig. 4. Proving that negation is an involution in Agda.

As a starting point, we take the Agda proof from Figure 4. In that proof, the Booleans are defined as a datatype, negation is defined as a non-dependent function from Booleans to Booleans, and `neg_inverse` is defined as a dependent function from a boolean x to a proof that x is equal to $\mathtt{neg}(\mathtt{neg}\ x)$. In our system, we want to express both `neg` and `neg_inverse` without using non-dependent or dependent functions. For the non-dependent case, we can express negation directly as an observation on Booleans:

```
def Bool.neg: Bool {
  True => False,
  False => True }
```

If we want to express the dependent function `neg_inverse` as an observation on Booleans in a similar way, then we have to add a feature, *self parameters*. We can bind the term that we observe to a variable, which we have here called *self*, and use this variable in the return type of the observation:

```
def (self: Bool).neg_inverse: Eq(Bool, self, self.neg.neg) {
  True => Refl(Bool, True),
  False => Refl(Bool, False) }
```

The refunctionalization of these methods with self-parameters dictates the first feature of dependent codata types: self-parameters in destructors.

```
codata Bool { neg: Bool, (self: Bool).neg_inverse: Eq(Bool, self, self.neg.neg) }
```

It is self-parameters in codata declarations which give us the expressive power to properly represent verified interfaces and classes in an object-oriented style.

2.2 Verified Interfaces and Classes

When verifying data structures and algorithms in dependently typed languages, we can choose between two general approaches: intrinsic and extrinsic verification. Using intrinsic verification, we define the data structure or algorithm together with its correctness proofs. To intrinsically verify data structures, we commonly express properties using type indices, such as the number of elements contained in a length-indexed list. We can employ a similar approach in an object-oriented style using indexed codata types [Thibodeau et al. 2016]. Using type indices, we can ensure

```
class spec: PR
    public methods:
      store: X × A ⟶ X
      read: X ⟶ {error} + A
      empty: X ⟶ X
    assertions:
      s.empty.read = error
      s.read = error
         ⊢ s.store(a).read = a
      s.read = a
         ⊢ s.store(b).read = a
    creation:
      new.read = error
  end class spec
```

(a) Original specification

```
codata PR {
   store(a: A): PR,
   read: MaybeA,
   empty: PR,
   -- | Reading from the empty buffer yields an error
   (s: PR).assert_empty: Eq(MaybeA, s.empty.read, Error),
   -- | We can store an element into an empty buffer
   (s: PR).assert_empty_store(a: A)
      : Eq(MaybeA, s.read, Error) -> Eq(MaybeA, s.store(a).read, Just(a)),
   -- | We cannot replace the element in the buffer without calling `empty`
   (s: PR).assert_persistent(a b: A)
      : Eq(MaybeA, s.read, Just(a)) -> Eq(MaybeA, s.store(b).read, Just(a)) }
```

(b) Implementation in our system.

Fig. 5. Persistent read (PR) specification for one-element buffers from Jacobs [1995].

that observations are called only on objects which are in the right state. This can be seen in the following example, where the observation `read` may be called only on non-empty buffers:[2]

```
codata Buffer(m: Nat) {
   Buffer(S(n)).read(n: Nat): Pair(Bool, Buffer(n)) }
codef EmptyBuffer: Buffer(Z) { read(n) absurd }
codef Singleton(b: Bool): Buffer(S(Z)) { read(n) => MkPair(Bool, Buffer(Z), b, EmptyBuffer) }
```

We can see that, as usual for dependent (co)pattern matching, infeasible pattern matches may arise which need to be marked as `absurd`. That is, when we implement the buffer interface for the empty buffer we don't have to implement the `read` method since it can never be called.

However, in this work, we go beyond indexed codata types, which admit an intrinsic verification style. We also want to support the extrinsic approach, where we want to separate our objects from their specifications. Jacobs [1995] provides us with an initial concept of how to attain that goal. He proposes a system of coalgebraic specifications that can be used to verify object-oriented classes. As an example, Figure 5a shows a coalgebraic specification for a one-element buffer that exhibits *persistent read* (PR) behavior: After an element has been stored, it cannot be replaced using the `store` method. Instead, one needs to call the method `empty` to explicitly empty the buffer. Reading from the empty buffer returns an error. This specification of the buffer is given as a set of assertions that reference the buffer state `s`.

In our system, we can realize this concept using self-parameters on destructors, allowing us to express specifications as observations on codata types. The codata type in Figure 5 defines the verified interface for persistent read buffers in our system. Similarly, we can apply this approach to express verified interfaces such as functors or monads.

2.3 Dependent Functions

Unlike in most other dependent type theories, the Π-type of dependent functions is not part of our core theory, but can be defined in a library. The Π-type is defined as a codata type indexed over a type family `p`, for which we use the ordinary non-dependent function type. The notation `a -> b` used in the definition of dependent functions is just syntactic sugar for `Fun(a,b)`.

```
-- | Non-dependent Functions
codata Fun(a b: Type) {
   Fun(a, b).ap(a b: Type, x: a): b }
-- | Dependent Functions
codata Π(a: Type, p: a -> Type) {
   Π(a, p).dap(a: Type, p: a -> Type, x: a): p.ap(a, Type, x) }
```

[2]Note that the parameter *n* occurs bound in the type `Buffer(S(n))` on which we can call the observation `read`, and is bound in the argument list `read(n: Nat)`.

We propose that both dependent and non-dependent functions should be user-defined instead of built-in. The designers of Java decided to follow this approach when they introduced lambda abstractions as instances of *functional interfaces* [Goetz et al. 2014; Setzer 2003] in Java 8. This shows that our proposal is not radical, and we think it is also useful. Apart from reducing the complexity of the core language, they simplify the situation if we have more than one function type. This is the case in substructural systems where we have linear and non-linear functions. For instance, in the Rust programming language, there are three different built-in function traits `Fn`, `FnOnce`, and `FnMut`, which differ in the modality of the receiver[3].

2.4 Weak and Strong Dependent Pairs

In the previous section we showed how to define the Π-type. For the Π-type we had no choice but to define it as a codata type[4], but for the Σ-type we can choose whether to model it as a data or codata type. This choice distinguishes weak and strong Σ-types [Howard 1980]: *Strong* Σ-types are defined as a codata type with two projections, where the second projection mentions the result of the first projection in its return type; *weak* Σ-types, by contrast, are defined as a data type with one constructor which pairs the first and second element. This difference is more obvious if we first consider the case of non-dependent pairs, which can also be written as either a data or codata type.

```
data ×₊(A B: Type) {
  Pair(A B: Type, x: A, y: B): ×₊(A, B) }
def ×₊(A, B).π₁(A B: Type): A {
  Pair(_, _, x, y) => x }
def ×₊(A, B).π₂(A B: Type): B {
  Pair(_, _, x, y) => y }
```

```
codata ×₋(A B: Type) {
  ×₋(A, B).π₁(A B: Type): A,
  ×₋(A, B).π₂(A B: Type): B }
codef Pair(A B: Type, x: A, y: B): ×₋(A, B) {
  π₁(_, _) => x,
  π₂(_, _) => y }
```

These two representations can be obtained from each other by defunctionalization and refunctionalization. This is still the case when we generalize non-dependent pairs to the Σ-type. Similar to the Π-type in Section 2.3, the Σ-type is indexed by a type family T. As a data type, it is defined by one constructor `Pair` which takes the type family T, an element x of type A and a witness w as arguments. As a codata type, we still have two projections π_1 and π_2 as in the case of non-dependent pairs. But the second projection now uses the self-parameter to guarantee that an element of type T applied to $self.\pi_1$ is returned.

```
data Σ₊(A: Type, T: A -> Type) {
  Pair(A: Type,
       T: A -> Type,
       x: A,
       w: T.ap(A, Type, x) )
     : Σ₊(A, T) }
def Σ₊(A, T).π₁(A: Type, T: A -> Type): A {
  Pair(A, T, x, w) => x }
def (self: Σ₊(A, T)).π₂(A: Type, T: A -> Type)
  : T.ap(A, Type, self.π₁(A, T)) {
  Pair(A, T, x, w) => w }
```

```
codata Σ₋(A: Type, T: A -> Type) {
  Σ₋(A, T).π₁(A: Type, T: A -> Type): A,
  (self: Σ₋(A, T)).π₂(A: Type, T: A -> Type)
    : T.ap(A, Type, self.π₁(A, T)) }
codef Pair(A: Type,
           T: A -> Type,
           x: A,
           w: T.ap(A, Type, x) )
  : Σ₋(A, T) {
  π₁(A, T) => x,
  π₂(A, T) => w }
```

In fact, Agda can already represent Σ-types in both of these ways. But there is one caveat: Agda was not originally designed with codata types in mind, and its codata types are implemented on top of dependent records, which limits what kind of codata types are possible. For example, the order of the destructors in a codata type matter for Agda, so we cannot reorder the first and second projection. In our system the destructors of a codata type are not ordered and can mutually refer to each other, which precisely mirrors how definitions are mutually recursive on the toplevel.

[3]See the Rust standard library documentation on operators: doc.rust-lang.org/std/ops/index.html.
[4]If we want to define the function type as a data type, then we have to use a system with higher-level inference rules, cf. Garner [2009].

But why should we care about these two alternative encodings of the Σ-type? Take, for example, Eisenberg et al. [2021] who discuss the addition of existential types to Haskell. Since Haskell both is lazy and supports type erasure, Eisenberg et al. are driven to a design that uses strong existential types. We think that by using the framework of data and codata types we can make these kind of differences even clearer.

2.5 Codata Encodings of Natural Numbers

Starting with the inception of the lambda calculus, researchers have been interested in *functional encodings* of data types such as booleans, natural numbers and lists. Classical examples of functional encodings are the Church, Scott and Parigot encodings of data types (cf. Geuvers [2014]; Koopman et al. [2014]). Functions are from our perspective just one particular instance of a codata type, so we are interested in the more general problem of *codata encodings* instead of functional encodings. Most codata encodings of data types can be obtained by refunctionalizing a data type with an appropriate observation. That the Church encoding can be obtained from refunctionalizing a program with Peano numbers and an `iter` function has already been observed by Ostermann and Jabs [2018]; we restate this example in Figure 6.

```
data Nat { Z, S(p: Nat) }
def Nat.iter(A: Type, z: A, s: A -> A): A {
  Z => z,
  S(p) => s.ap(A, A, p.iter(A, z, s)) }
```

(a) Data variant

```
codata Nat { iter(A: Type, z: A, s: A -> A): A }
codef S(p: Nat): Nat {
  iter(A, z, s) => s.ap(A, A, p.iter(A, z, s)) }
codef Z: Nat { iter(A, z, s) => z }
```

(b) Codata variant

Fig. 6. The Church encoding as a refunctionalized program on Peano numbers.

We can observe that the codata type in Figure 6b which represents the Church encoding of natural numbers is not recursive. This corresponds to the well-known theorem that Church encodings can be typed in pure system F. If we apply the same method to obtain the Scott or Parigot encoding of natural numbers, then we can observe that the resulting codata type is recursive. This corresponds to the other well-known theorem that these encodings can not be typed in pure System F and require recursive types.

We can even go one step further. Geuvers [2001] showed that these previous encodings cannot express induction or dependent elimination. One way to obtain typed functional encodings which can express induction is to add a form of self types to the system; this kind of encoding was introduced by Fu and Stump [2014]. While it is hard to prove an exact correspondence, we think that the essential idea of the encoding of Fu and Stump can be expressed in our system in Figure 7 and Figure 8.

```
codef StepFun(P: Nat -> Type): Fun(Nat, Type) {
  ap(_, _, x) => P.ap(Nat, Type, x) -> P.ap(Nat, Type, S(x)) }
data Nat { S(m: Nat), Z }
def (n: Nat).ind(P: Nat -> Type, base: P.ap(Nat, Type, Z), step: Π(Nat, StepFun(P)))
  : P.ap(Nat, Type, n) {
  S(m) =>
    step.dap(Nat, StepFun(P), m)
        .ap(P.ap(Nat, Type, m), P.ap(Nat, Type, S(m)), m.ind(P, base, step)),
  Z => base }
```

Fig. 7. The data type of natural numbers with an induction principle.

In Figure 7 we have encoded induction using a helper codata type `StepFun` which encodes the induction step for a given predicate P on natural numbers. Induction is then expressed as the observation `ind` on a natural number n which expects the base case and the induction step of the

induction as arguments. The argument n on which we define the observation occurs itself in the return type. Refunctionalization of this program results in Figure 8.

```
codef StepFun(P: Nat -> Type): Fun(Nat, Type) {
   ap(_, _, x) => P.ap(Nat, Type, x) -> P.ap(Nat, Type, S(x)) }
codata Nat {
   (n: Nat).ind(P: Nat -> Type, base: P.ap(Nat, Type, Z), step: Π(Nat, StepFun(P)))
      : P.ap(Nat, Type, n) }
codef Z: Nat { ind(P, base, step) => base }
codef S(m: Nat): Nat {
   ind(P, base, step) =>
      step.dap(Nat, StepFun(P), m)
          .ap(P.ap(Nat, Type, m), P.ap(Nat, Type, S(m)), m.ind(P, base, step)) }
```

Fig. 8. The encoding of Fu and Stump can be obtained by refunctionalizing the program in Figure 7.

We think that this is further evidence that the self-parameters we introduced to the system occur naturally when we go from the non-dependent to the dependent setting.

3 CASE STUDY

We will now further illustrate the benefits of dependently typed object-oriented programming in a small case study. For this, we create a mockup of a dependently typed web server. We will observe that we can conveniently extend both the supported routes of the web server and the supported methods to access these routes. We will also see how we can conveniently state and enforce properties in intrinsic as well as in extrinsic style.

3.1 A Functional Web Server

We start in the familiar realm of functional programming. For the purpose of this demonstration, we will create a simple web server that allows all users to read, but only authenticated users to increment a counter. For this, we track user sessions using the `State` type shown below. As an instance of intrinsic verification, we track on the type level whether the user is authenticated. Possible responses from the server are specified by the `Response` type.

```
codata User { hasCredentials: Bool }
codata State(loggedIn: Bool) {
   State(False).login(u: User): State(u.hasCredentials),
   State(True).logout: State(False),
   State(True).increment: State(True),
   State(True).set(n: Nat): State(True),
   State(b).counter(b: Bool): Nat }
data Response { Forbidden, Return(n: Nat) }
```

Our web server should accept a couple of HTTP request methods (`get`, `post`, ...) for a set of routes (`Index`, `Admin`, ...).

```
data Route { Index }
def Route.requiresLogin: Bool { Index => False }
def (self: Route).get: State(self.requiresLogin) -> Response {
   Index => \state. Return(state.counter(False)) }
```

Adding support for a new request method is as simple as adding a function. For instance, we want to handle post requests, even though we forbid them for the `Index` route:

```
def (self: Route).post: State(self.requiresLogin) -> ×_(State(self.requiresLogin), Response) {
   Index =>
      \state. comatch {
         fst(a, b) => state,
         snd(a, b) => Forbidden } }
```

3.2 Adding New Routes in Object-Oriented Style

While adding new methods is a local change, adding a new route in the functional representation requires touching all pattern matches on `Route` in the program. Therefore, before adding a route

to increment the counter on a `POST` request, let us refunctionalize `Route` to its object-oriented decomposition:

```
codata Route {
   requiresLogin: Bool,
   (self: Route).get: State(self.requiresLogin) -> Response,
   (self: Route).post: State(self.requiresLogin) -> ×₋(State(self.requiresLogin), Response) }
codef Index: Route {
   requiresLogin => False,
   get => \state. Return(state.counter(False)),
   post =>
      \state. comatch {
         fst(a, b) => state,
         snd(a, b) => Forbidden } }
```

In the object-oriented decomposition, adding the following `Admin` route is now a local change. Note that the rearrangement works both ways: we could transpose the program back into functional decomposition to add another method.

```
codef Admin: Route {
   requiresLogin => True,
   post =>
      \state. comatch {
         fst(a, b) => state.increment,
         snd(a, b) => Return(state.increment.counter(True)) },
   get => \state. Return(state.counter(True)) }
```

Similar problems of modularity appear in many applications. Functional languages force us to always choose the same extensibility dimension for every type: We can extend data types with new observations, but we cannot easily extend types with new constructors. If the programming language would support both programming paradigms equally well, this choice would not be forced on the programmer by the language, but the programmer would have the choice for each type.

3.3 Verifying Properties on Routes

In addition to the ability to increment the counter by sending a `POST` request to the `Admin` route, we may also want to allow explicitly setting a counter value. Updating the counter to a value should be idempotent, i.e. calling the route more than once should have the same effect. The HTTP `PUT` method is supposed to capture this behavior, but how can we enforce it in our code? This leads us to another benefit of the object-oriented style in that we can express such properties extrinsically but still as part of the interface (compare 2.2):

```
data Utils { MkUtils }
def Utils.put_twice(route: Route, request: Request, state: State): Pair(State, Response) {
   MkUtils => route.put(request, route.put(request, state).fst(State, Response)) }
codata Route {
   (self: Route).put(request: Request, state: State): Pair(State, Response),
   (self: Route).put_idempotent(request: Request, state: State)
      : Eq(Pair(State, Response), self.put(request, state), MkUtils.put_twice(self, request, state)) }
```

The full code of the case study with the added `put` method is contained in the artifact Binder et al. [2024a].

4 DESIGN CONSTRAINTS AND SOLUTIONS

Specifying a consistent set of typing and computation rules for data and codata types is not difficult. In this section, we show the difficulties that arise if we also want the rules to be closed under defunctionalization and refunctionalization. That is every program that typechecks should continue to typecheck if we defunctionalize or refunctionalize any of the types that occur in it.

4.1 Judgmental Equality of Comatches

Problem. For most dependently typed languages, the term $\lambda x.x$ is judgmentally equal to the term $\lambda y.y$, and likewise $\lambda x.2 + 2$ and $\lambda x.4$ are considered equal. Equating such terms becomes a problem, however, if we want to defunctionalize the programs which contain them. Different lambda abstractions in a program are defunctionalized to different constructors, which are then no longer judgmentally equal. Let us illustrate the problem with an example.

Consider the following proof that $\lambda y.y$ is the same function from natural numbers to natural numbers as $\lambda z.z$. We prove this fact using a third lambda abstraction $\lambda x.x$ as an argument to the reflexivity constructor.

```
codata Fun(a b: Type) { Fun(a, b).ap(a b: Type, x: a): b }
Refl(Fun(Nat, Nat), \x. x) : Eq(Fun(Nat, Nat), \y. y, \z. z)
```

If we defunctionalize this program, then each of these three lambda abstractions becomes one constructor of the data type. However since different constructors are not judgmentally equal, the following defunctionalized program no longer typechecks.

```
data Fun(a b: Type) { F1: Fun(Nat, Nat), F2: Fun(Nat, Nat), F3: Fun(Nat, Nat) }
def Fun(a, b).ap(a b: Type, x: a): b { F1 => x, F2 => x, F3 => x }
Refl(Fun(Nat, Nat), F1) : Eq(Fun(Nat, Nat), F2, F3)
```

Here is the gist of the problem: *Judgmental equality must be preserved by defunctionalization and refunctionalization.* This means that if we don't want to treat different constructors of a data type as judgmentally equal, then *we cannot treat all α-β-equivalent comatches as judgmentally equal* either.

It is not impossible to devise a scheme which lifts judgmentally equal comatches to the same constructors. However, we decided against this as it leads to confusing behavior. First, de- and refunctionalization would no longer be inverse transformations at least under syntactic equality. Second, such an attempt would necessarily be a conservative approximation as program equivalence is undecidable in general. In practice, that would mean that some comatches would be collapsed to the same constructor during lifting, while others would not.

Solution. Note that the opposite approach—never equating any comatches—doesn't work either, since typing would then no longer be closed under substitution. For example, if f is a variable standing for a function from natural numbers to natural numbers, then the term $\mathrm{Refl}(\mathrm{Fun}(\mathrm{Nat}, \mathrm{Nat}), f)$ is a proof of the proposition $\mathrm{Eq}(\mathrm{Fun}(\mathrm{Nat}, \mathrm{Nat}), f, f)$. But we could not substitute a comatch $\lambda y.y$ for f, since the result would no longer typecheck. We therefore have to find a solution between these two extremes.

Our solution consists of always considering local comatches together with a name[5]. Only comatches which have the same name are judgmentally equal, and this equality is preserved by reduction since the comatch is duplicated together with its name.

Where do the names for local comatches come from? We support user-annotated labels, which allow the programmer to give meaningful names to comatches. Manually naming comatches in this way is useful as these labels can also be used by defunctionalization to name the generated constructors. We enforce that these user-annotated labels are globally unique. However, as we do not want to burden the user with naming every single comatch in the program, we also allow unannotated comatches, for which we automatically generate unique names. As a result, each comatch occurring textually in the program has a unique name, but these names possibly become duplicated during normalization and typechecking.

[5]This solution is similar to Binder et al.'s use of labels for local (co)pattern matches

4.2 Eta Equality

Problem. For reasons very similar to the previous section, η-equality is not preserved under defunctionalization and refunctionalization. Let us again consider a simple example. In the following proof, we show that a function f is equal to its η-expanded form $\lambda x.f.\texttt{ap}(x)$. In order to typecheck, the proof would need to use a judgmental η-equality for functions.

```
codata Fun { ap(x: Nat): Nat }
let prop_eta(f: Fun): Eq(Fun, f, (\x. f.ap(x))) := Refl(Fun, f);
```

However, defunctionalization of this proof would result in the following program, where we have used an ellipsis to mark all the constructors that were generated for the other lambda abstractions in the program.

```
data Fun { Eta(f: Fun), ... }
def Fun.ap(x: Nat): Nat { Eta(f) => f.ap(x),... }
let prop_eta(f: Fun): Eq(Fun, f, Eta(f)) := Refl(Fun, f);
```

Using `prop_eta` it would now be possible to show that any constructor `f` of `Fun` is equal to `Eta(f)`. This would contradict the provable proposition that distinct constructors are not equal.

Solution. We do not support η-equality in our formalization and implementation. This means that we only normalize β-redexes but not η-redexes during typechecking. However, it would be possible to support judgmental η-equality on a case-by-case basis similar to the `eta-equality` and `no-eta-equality` keywords in Agda which enable or disable `eta`-equality for a specific record type[6]. De- and refunctionalization is then only available for types without η-equality.

5 FORMALIZATION

In this section, we present the syntax, typing rules and operational semantics of our system. We divide this presentation into three subsections: In Section 5.1, we introduce the core of our system. We extend this core calculus by data types and pattern matching definitions in Section 5.2, and by codata types and copattern matching definitions in Section 5.3.

We do not formalize local pattern and copattern matches. Instead, local pattern and copattern matches are lifted to the top level before applying de- or refunctionalization, similar to the approach taken by Binder et al. [2019]. Some care must be taken to ensure that we close over all required terms, as the types of terms which are part of the closure might close over additional terms. For example, closing over `v: Vec n` requires us to also close over `n`. The main challenge for local pattern and copattern matches revolves around judgmental equality, which we discussed in Section 4.1.

5.1 Core System

In Figure 9 we define the syntax of our core system together with small examples in the rightmost column.

Following standard convention, we formalize our system up to α-renaming of bound variables x, y, z. We distinguish between contexts Γ, Δ and telescopes Ξ, Ψ. Contexts track the types of free variables and must always be closed. Telescopes are dependent parameter lists whose types may contain free variables bound in a context. If a telescope is closed, we may implicitly use it as a context. A substitution ρ, σ is an argument list to a telescope. A program Θ is a list of declarations δ, which are empty for now. There are five different kinds of expressions e, s, t: Variables are denoted as described above. We denote the type universe as Type. Type constructors $\mathcal{T}\rho$ instantiate a (co)data type with a substitution ρ. Calling a producer C is written Cσ; invoking a consumer d uses the syntax e.dσ. The producer syntax denotes constructor calls for data types and codefinition calls

[6] Compare the section on record types in the Agda user manual: agda.readthedocs.io/en/v2.6.3/language/record-types.html.

$\mathcal{T}$	$\in$	TypeNames	*Type names*	`Bool`, `Vec`, `Stream`
C	$\in$	ProducerNames	*Producer names*	`True`, `Cons`
d	$\in$	ConsumerNames	*Consumer names*	`neg`, `neg_inverse`
x, y, z	$\in$	Variables	*Variables*	
δ	::=	(empty in core system)	*Declaration*	
Θ	::=	$\emptyset \mid \delta, \Theta$	*Program*	
Γ, Δ	::=	$\emptyset \mid \Gamma, \mathrm{x} : t$	*Context*	$\mathrm{x} : \texttt{Nat}, \mathrm{v} : \texttt{Vec}(\texttt{Bool}, \mathrm{x})$
Ξ, Ψ	::=	$\emptyset \mid \mathrm{x} : t, \Xi$	*Telescope*	$\mathrm{x} : \texttt{Nat}, \mathrm{v} : \texttt{Vec}(\texttt{Bool}, \mathrm{x})$
ρ, σ	::=	$() \mid (e, \sigma)$	*Substitution*	$(\texttt{Bool}, \texttt{S}(\texttt{Z}))$
e, s, t	::=	x	*Variable*	
	$\mid$	Type	*Universe*	
	$\mid$	$\mathcal{T}\rho$	*Type*	$\texttt{Bool}, \texttt{Vec}(\texttt{Bool}, \texttt{S}(\texttt{Z}))$
	$\mid$	$\mathrm{C}\sigma$	*Producer*	$\texttt{S}(\texttt{Z})$
	$\mid$	$e.\mathrm{d}\sigma$	*Consumer*	$\mathrm{x}.\texttt{neg}$

Fig. 9. Syntax of core system without data or codata types.

for codata types. The consumer syntax denotes destructor invocations for codata and definition calls for data types.

For formalizing the core system, we closely follow the presentation by Hofmann [1997]. The rules for contexts, telescopes, and substitutions are standard, and we omit them here for space reasons. We will use the judgment forms $\vdash_\Theta \Gamma$ ctx and $\Gamma \vdash_\Theta \Xi$ tel to specify valid contexts and telescopes, respectively. The judgment form $\Gamma \vdash_\Theta \sigma : \Xi$ states that σ is a valid substitution for the telescope Ξ under context Γ.

We also assume the Type-in-Type axiom, which is well-known to be inconsistent [Girard 1972; Hurkens 1995]. In this work, we do not enforce termination or productivity in our system. As adding the Type-in-Type axiom merely yields another source of possible divergence [Tennant 1982], we do not gain anything from avoiding this paradox, e.g. by using a hierarchy of universes. We therefore follow Eisenberg [2016] and opt for a simpler presentation using the Type-in-Type axiom.

5.2 Data Types and Dependent Pattern Matching

We now extend the declarations of Figure 9 by two new constructs: data type declarations and pattern matching definitions. Data type declarations introduce a new data type together with a list of constructors, and pattern matching definitions introduce a top-level consumer which is defined by a list of clauses. The corresponding new typing and well-formedness rules are contained in the upper half of Figure 10.

δ	::=	...	*Extends Figure 9*	
	$\mid$	**data** $\mathcal{T}\Psi\ \{\ \overline{\mathrm{C}\Xi : \mathcal{T}\rho}\ \}$	*Data Type*	**data** `Bool` { `True` : `Bool`, ... }
	$\mid$	**def** $(\mathrm{z} : \mathcal{T}\rho).\mathrm{d}\Xi : t\ \{\ \overline{a}\ \}$	*Pattern Match*	**def** $(\mathrm{x} : \texttt{Bool}).\texttt{neg} : \texttt{Bool}\ \{\ldots\}$
a	::=	$\mathrm{C}\ \Xi \mapsto e$	*Match case*	$\texttt{S}(\mathrm{x} : \texttt{Nat}) \mapsto \mathrm{x}$
	$\mid$	$\mathrm{C}\ \Xi$ **absurd**	*Absurd case*	`Nil` (a : Type) **absurd**

A data type declaration **data** $\mathcal{T}\Psi\ \{\ \overline{\mathrm{C}\Xi : \mathcal{T}\rho}\ \}$ introduces one type constructor $\mathcal{T}$ and a list of term constructors C to the rest of the program. The type constructor $\mathcal{T}$ is indexed by a telescope Ψ, and if we provide a substitution ρ for this telescope, then the formation rule F-Data allows us

to form the type $\mathcal{T}\rho$. Each term constructor is declared with parameters Ξ and a type $\mathcal{T}\rho$ which specifies how the term constructor instantiates the indices of the type constructor. We can use a term constructor with the introduction rule I-Data, where the resulting type depends on both the arguments σ to which the term constructor is applied and the substitution ρ from the constructor declaration. We check that a data type declaration is well-formed with the help of the rule Data: For the type constructor $\mathcal{T}$ we check that the indices Ψ form a valid telescope in the program, and for each constructor $\mathrm{C}\Xi : \mathcal{T}\rho$ we check that Ξ is a valid telescope, and that ρ is a valid substitution for the indices Ψ of the type constructor $\mathcal{T}$ under context Ξ.

The second new construct is pattern matching definitions $\mathbf{def}\ (\mathrm{z} : \mathcal{T}\rho).\mathrm{d}\Xi : t\ \{\ \overline{a}\ \}$ which define a new consumer d by a list of cases. The consumer d takes arguments specified by the telescope Ξ, and can be called on any term of type $\mathcal{T}\rho$, where the substitution ρ can depend on any arguments bound in Ξ. The scrutinee of the consumer can be referred to by the self parameter z in the return type t of d. We introduced these self-parameters in Section 2.1. The elimination rule E-Data then eliminates a term e by invoking d with arguments σ. These arguments are substituted in the return type t which is defined under the context $\Xi; z : \mathcal{T}\rho$. Here, the self parameter z is replaced by the scrutinee e.

We check whether a pattern matching definition is well-formed with the rule Def. The return type t is typed under context $\Xi; \mathrm{z} : \mathcal{T}\rho$. We implicitly require that there exists exactly one case of each constructor C of $\mathcal{T}$, and check that every case is well-formed. For checking the well-formedness of cases we use an auxiliary judgment form $\Xi \vdash_\Theta a_i : (z : \mathcal{T}\rho).t$ which tracks the self parameter $\mathrm{z} : \mathcal{T}\rho$ and the return type t. There are two variants of pattern matching cases we have to consider: possible and absurd cases. In a possible case, we restate the constructor telescope Ξ and give an expression e that gives the result if the case is matched. An absurd case is determined to be impossible by unification and hence does not need an expression. Corresponding to the two kinds of cases, possible and absurd, there are two typing rules, $\textsc{Case}_1$ and $\textsc{Case}_2$. In both rules, we unify the scrutinee type $\mathcal{T}\rho_1$ with the constructor type $\mathcal{T}\rho_2$. If there is no such unifier, the case is absurd. In a possible case, however, unification yields a unifier θ. We use this unifier to refine the typing of the right-hand side e: The unifier θ is substituted in the context $\Xi_1; \Xi_2$, expression e and type t. We further refine t by replacing the self parameter z with the constructor $\mathrm{C}\mathrm{id}_{\Xi_2}$ where id_{Ξ_2} is the identity context morphism for Ξ_2.

Finally, the computation rule ≡-Data allows us to reduce an expression $\mathrm{C}\sigma_1.\mathrm{d}\sigma_2$ if σ_1 and σ_2 are valid arguments to the constructor respectively definition.

5.3 Codata Types and Dependent Copattern Matching

Finally, we extend programs by codata type declarations and copattern matching definitions. Codata type declarations introduce a new codata type together with a list of destructors, and copattern matching definitions introduce a new top-level producer by a list of copattern matching clauses. Their typing and well-formedness rules are contained in the lower half of Figure 10.

$$
\begin{array}{rcll}
\delta & ::= & \ldots & \textit{Extends Figure 9} \\
 & | & \mathbf{codata}\ \mathcal{T}\Psi\ \{\ \overline{(\mathrm{z} : \mathcal{T}\rho).\mathrm{d}\Xi : t}\ \} & \textit{Codata declaration} \\
 & | & \mathbf{codef}\ \mathrm{C}\Xi : \mathcal{T}\rho\ \{\ \overline{o}\ \} & \textit{Producer declaration} \\
o & ::= & \mathrm{d}\ \Xi \mapsto e & \textit{Possible cocase} \\
 & | & \mathrm{d}\ \Xi\ \mathbf{absurd} & \textit{Absurd cocase}
\end{array}
$$

A codata type declaration $\mathbf{codata}\ \mathcal{T}\Psi\ \{\ \overline{(\mathrm{z} : \mathcal{T}\rho).\mathrm{d}\Xi : t}\ \}$ introduces the type constructor $\mathcal{T}$ and a list of destructors d to the program. Like data types, codata types are indexed, and the type constructor $\mathcal{T}$ can be instantiated to form types with the help of the rule F-Codata. The signature

Declaration Rules

$$\frac{\vdash_\Theta \Psi \text{ tel} \quad \forall i : \left[\vdash_\Theta \Xi_i \text{ tel and } \Xi_i \vdash_\Theta \rho_i : \Psi \right]}{\vdash_\Theta \textbf{data } \mathcal{T}\Psi \ \{ \overline{\mathrm{C}\Xi : \mathcal{T}\rho} \} \ \text{Ok}} \textsc{Data} \qquad \frac{\textbf{data } \mathcal{T}\Psi \ \{...\} \in \Theta \quad \Xi; \mathrm{z} : \mathcal{T}\rho \vdash_\Theta t : \text{Type} \quad \Xi \vdash_\Theta \rho : \Psi \quad \forall i : \ \Xi \vdash_\Theta a_i : (\mathrm{z} : \mathcal{T}\rho).t}{\vdash_\Theta \textbf{def } (\mathrm{z} : \mathcal{T}\rho).\mathrm{d}\Xi : t \ \{ \overline{a} \} \ \text{Ok}} \textsc{Def}$$

$$\frac{\textbf{data } \mathcal{T}\Psi \ \{ \mathrm{C}\Xi_2 : \mathcal{T}\rho_2,... \} \in \Theta \quad \Xi_1; \Xi_2 \vdash_\Theta \theta \text{ mgu for } \rho_1 \equiv \rho_2 : \Psi \quad (\Xi_1; \Xi_2)[\theta] \vdash_\Theta e[\theta] : t[\mathrm{C}\,\mathrm{id}_{\Xi_2}/\mathrm{z}][\theta]}{\Xi_1 \vdash_\Theta \mathrm{C}\ \Xi_2 \mapsto e : (\mathrm{z} : \mathcal{T}\rho_1).t} \textsc{Case}_1$$

$$\frac{\textbf{data } \mathcal{T}\Psi \ \{ \mathrm{C}\Xi_2 : \mathcal{T}\rho_2,... \} \in \Theta \quad \neg\exists \theta : \ \Xi_1; \Xi_2 \vdash_\Theta \theta \text{ mgu for } \rho_1 \equiv \rho_2 : \Psi}{\Xi_1 \vdash_\Theta \mathrm{C}\ \Xi_2 \ \textbf{absurd} : (\mathrm{z} : \mathcal{T}\rho_1).t} \textsc{Case}_2$$

Formation, Introduction, Elimination and Computation Rules

$$\frac{\textbf{data } \mathcal{T}\Psi \ \{...\} \in \Theta \quad \Gamma \vdash_\Theta \rho : \Psi}{\Gamma \vdash_\Theta \mathcal{T}\rho : \text{Type}} \text{F-Data} \qquad \frac{\textbf{data } \mathcal{T}\Psi \ \{ \mathrm{C}\Xi : \mathcal{T}\rho,... \} \in \Theta \quad \Gamma \vdash_\Theta \sigma : \Xi}{\Gamma \vdash_\Theta \mathrm{C}\sigma : \mathcal{T}\rho[\sigma/\Xi]} \text{I-Data}$$

$$\frac{\textbf{def } (z : \mathcal{T}\rho).\mathrm{d}\Xi : t \ \{...\} \in \Theta \quad \Gamma \vdash_\Theta \sigma : \Xi \quad \Gamma \vdash_\Theta e : \mathcal{T}\rho[\sigma/\Xi]}{\Gamma \vdash_\Theta e.\mathrm{d}\sigma : t[\sigma/\Xi][e/\mathrm{z}]} \text{E-Data}$$

$$\frac{\textbf{data } \mathcal{T}\Psi \ \{ \mathrm{C}\Xi_1 : \mathcal{T}\rho_1,... \} \in \Theta \quad \textbf{def } (\mathrm{z} : \mathcal{T}\ \rho_2).\mathrm{d}\ \Xi_2 : t \ \{ \ \mathrm{C}\ \Xi_1 \mapsto e,... \ \} \in \Theta \quad \Gamma \vdash_\Theta \sigma_1 : \Xi_1 \quad \Gamma \vdash_\Theta \sigma_2 : \Xi_2 \quad \Gamma \vdash_\Theta \rho_1[\sigma_1/\Xi_1] \equiv \rho_2[\sigma_2/\Xi_2] : \Psi}{\Gamma \vdash_\Theta \mathrm{C}\sigma_1.\mathrm{d}\sigma_2 \equiv e[\sigma_2/\Xi_2][\sigma_1/\Xi_1] : t[\sigma_2/\Xi_2][\mathrm{C}\sigma_1/\mathrm{z}]} \equiv\text{-Data}$$

Declaration Rules

$$\frac{\vdash_\Theta \Psi \text{ tel} \qquad \forall i : \left[\begin{array}{c} \vdash_\Theta \Xi_i \text{ tel and} \\ \Xi_i \vdash_\Theta \rho_i : \Psi \text{ and} \\ \Xi_i; \mathrm{z}_i : \mathcal{T}\rho_i \vdash_\Theta t_i : \text{Type} \end{array} \right]}{\vdash_\Theta \textbf{codata } \mathcal{T}\ \Psi \ \{ \overline{(\mathrm{z} : \mathcal{T}\rho).\ \mathrm{d}\ \Xi : t} \} \ \text{Ok}} \textsc{Codata} \qquad \frac{\textbf{codata } \mathcal{T}\Psi\{...\} \in \Theta \quad \Xi \vdash_\Theta \rho : \Psi \quad \forall i, \Xi \vdash_\Theta o_i : (C : \mathcal{T}\rho)}{\vdash_\Theta \textbf{codef } \mathrm{C}\ \Xi : \mathcal{T}\ \rho \ \{ \ \overline{o} \ \} \ \text{Ok}} \textsc{Codef}$$

$$\frac{\textbf{codata } \mathcal{T}\ \Psi \ \{ \ (\mathrm{z} : \mathcal{T}\rho_2).\ \mathrm{d}\ \Xi_2 : t \ \} \in \Theta \quad \Xi_1; \Xi_2 \vdash_\Theta \theta \text{ mgu for } \rho_1 \equiv \rho_2 : \Psi \quad (\Xi_1; \Xi_2)[\theta] \vdash_\Theta e[\theta] : t[\mathrm{C}\ \mathrm{id}_{\Xi_1}/\mathrm{z}][\theta]}{\Xi_1 \vdash_\Theta \mathrm{d}\ \Xi_2 \mapsto e : (C : \mathcal{T}\ \rho_1)} \textsc{Cocase}_1$$

$$\frac{\textbf{codata } \mathcal{T}\ \Psi \ \{ \ (\mathrm{z} : \mathcal{T}\rho_2).\ \mathrm{d}\ \Xi_2 : t \ \} \in \Theta \quad \neg\exists \theta, \Xi_1; \Xi_2 \vdash_\Theta \theta \text{ mgu for } \rho_1 \equiv \rho_2 : \Psi}{\Xi_1 \vdash_\Theta \mathrm{d}\ \Xi_2 \ \textbf{absurd} : (C : \mathcal{T}\ \rho)} \textsc{Cocase}_2$$

Formation, Introduction, Elimination and Computation Rules

$$\frac{\textbf{codata } \mathcal{T}\Psi \ \{...\} \in \Theta \quad \Gamma \vdash_\Theta \rho : \Psi}{\Gamma \vdash_\Theta \mathcal{T}\rho : \text{Type}} \text{F-Codata} \qquad \frac{\textbf{codef } \mathrm{C}\Xi : \mathcal{T}\rho \ \{...\} \in \Theta \quad \Gamma \vdash_\Theta \sigma : \Xi}{\Gamma \vdash_\Theta \mathrm{C}\sigma : \mathcal{T}\rho[\sigma/\Xi]} \text{I-Codata}$$

$$\frac{\textbf{codata } \mathcal{T}\Psi \ \{(\mathrm{z} : \mathcal{T}\rho).\mathrm{d}\Xi : t,...\} \in \Theta \quad \Gamma \vdash_\Theta \sigma : \Xi \quad \Gamma \vdash_\Theta e : \mathcal{T}\rho[\sigma/\Xi]}{\Gamma \vdash_\Theta e.\mathrm{d}\sigma : t[\sigma/\Xi][e/\mathrm{z}]} \text{E-Codata}$$

$$\frac{\textbf{codata } \mathcal{T}\Psi \ \{(z : \mathcal{T}\rho_2).\mathrm{d}\Xi_2 : t,...\} \in \Theta \quad \Gamma \vdash_\Theta \sigma_1 : \Xi_1 \quad \textbf{codef } \mathrm{C}\Xi_1 : \mathcal{T}\rho_1 \ \{\mathrm{d}\ \Xi_2 \mapsto e,...\} \in \Theta \quad \Gamma \vdash_\Theta \sigma_2 : \Xi_2 \quad \Gamma \vdash_\Theta \rho_1[\sigma_1/\Xi_1] \equiv \rho_2[\sigma_2/\Xi_2] : \Psi}{\Gamma \vdash_\Theta \mathrm{C}\sigma_1.\mathrm{d}\sigma_2 \equiv e[\sigma_1/\Xi_1][\sigma_2/\Xi_2] : t[\sigma_2/\Xi_2][\mathrm{C}\sigma_1/\mathrm{z}]} \equiv\text{-Codata}$$

Fig. 10. Well-formedness and typing rules for data and codata types.

of each destructor declaration $(z : \mathcal{T}\rho).\mathrm{d}\Xi : t$ corresponds precisely to the signature of a pattern matching definition for data types: Destructors take a parameter telescope Ξ and a self parameter $(z : \mathcal{T}\rho)$, where ρ is a substitution for the type parameters Ψ under context Ξ. The return type t is defined under the context $\Xi; z : \mathcal{T}\rho$. Using the rule E-Codata, we can eliminate a term e by calling a destructor d with arguments σ. In order to obtain the type of $e.\mathrm{d}\sigma$, we substitute σ for the parameters Ξ in the return type t, and e for the self parameter z. We use the rule Codata to check that a codata type declaration is well-formed.

A codefinition **codef** $\mathrm{C}\Xi : \mathcal{T}\rho\ \{\,\overline{o}\,\}$ has a signature which reflects the signature of constructors of a data type. The body of a codefinition is a list of *cocases*, which can be either possible or absurd. In a possible cocase, we restate the telescope Ξ and give an expression e which provides the result if the cocase gets matched. An absurd cocase is determined to be impossible by unification and hence does not need to provide an expression.

We check whether codefinitions are well-formed with the help of the rule Codef: We check that the parameters Ξ are valid and that the arguments ρ to the type constructor are well-typed under context Ξ. Further, we ensure that all cocases o_i are well-formed with the auxiliary judgment form $\Xi \vdash_\Theta m_i^d : (C : \mathcal{T}\ \rho)$, which tracks label C and type $\mathcal{T}\ \rho$ of the codefinition. We implicitly require that there exists one cocase for each destructor d of $\mathcal{T}$. As with pattern matching cases for data types, there are possible and absurd cocases as specified by the rules $\textsc{Cocase}_1$ and $\textsc{Cocase}_2$. Which rule applies is determined by unifying the codefinition type $\mathcal{T}\rho_1$ with the destructor type $\mathcal{T}\rho_2$. If no such unifier exists, the cocase is absurd. In a possible cocase, the unifier θ is substituted in context $\Xi_1; \Xi_2$, expression e, and type t. We also refine the return type t by substituting $\mathrm{C}\ \mathrm{id}_{\Xi_1}$, where id_{Ξ_1} is the identity context morphism for Ξ_1. Reminiscent of constructor calls, I-Codata introduces a term of such a type by invoking a codefinition C with arguments σ for the parameters Ξ. As these parameters Ξ may occur in the constructed type $\mathcal{T}\ \rho$, we substitute the arguments σ in the type.

Lastly, the computation rule ≡-Codata reduces a redex $\mathrm{C}\sigma_1.\mathrm{d}\sigma_2$ if σ_1 is a valid argument for the codefinition C and σ_2 is a valid argument for the destructor d.

5.4 Call-By-Value Operational Semantics

The computation rules of Section 5.2 and Section 5.3 do not specify a deterministic evaluation order. In this section we present the call-by-value (CBV) operational semantics of our system; the operational semantics does not depend on typing information. We specify evaluation by means of evaluation contexts in the style of Felleisen and Hieb [1992]. Values consist of the type universe or of type constructors and named producers applied to other values. Named producers can either be the constructors of a data type or the call to a toplevel codefinition. Such codefinitions generalize lambda abstractions which are lifted to the toplevel.

$$\begin{array}{lcl} v & ::= & \mathrm{Type} \mid \mathcal{T}\,\overline{v} \mid \mathrm{C}\,\overline{v} \\ E & ::= & \Box \mid \mathrm{C}\,\overline{v} E\,\overline{e} \mid \mathcal{T}\,\overline{v} E\,\overline{e} \mid E.\mathrm{d}\sigma \mid v.d\,\overline{v} E\,\overline{e} \end{array}$$

Reduction $e \triangleright e'$ happens when introduction and elimination forms meet, i.e. when a method is called on a constructor, or when a destructor is invoked on a codefinition:

$$\frac{e \triangleright_\beta e'}{E[e] \triangleright E[e']} \qquad \frac{\textbf{def}\ (z : \mathcal{T}\rho).\mathrm{d}\Xi_2 : t\ \{\mathrm{C}\ \Xi_1 \mapsto e\} \in \Theta}{\mathrm{C}\overline{v_1}.\mathrm{d}\overline{v_2} \triangleright_\beta e[\overline{v_2}/\Xi_2][\overline{v_1}/\Xi_1]} \qquad \frac{\textbf{codef}\ \mathrm{C}\Xi_1 : \mathcal{T}\rho_1\ \{\mathrm{d}\ \Xi_2 \mapsto e\} \in \Theta}{\mathrm{C}\overline{v_1}.\mathrm{d}\overline{v_2} \triangleright_\beta e[\overline{v_1}/\Xi_1][\overline{v_2}/\Xi_2]}$$

Here, $\triangleright_\beta$ denotes single step reduction of a direct redex, while $\triangleright$ denotes evaluation within an evaluation context.

5.5 Type Soundness

We show type soundness with respect to the call-by-value semantics of Section 5.4 by the usual progress and preservation theorems.

Theorem 5.1 (Progress). *For any well-formed program Θ and expressions e and t, if $\vdash_\Theta e : t$ then either e is a value or there exists an expression e' such that $e \triangleright e'$.*

Proof. Available in the online appendix and the extended version [Binder et al. 2024b]. □

Theorem 5.2 (Preservation). *For any well-formed program Θ and any expressions e_1, e_2, t if $\vdash_\Theta e_1 : t$ and $e_1 \triangleright e_2$, then $\vdash_\Theta e_2 : t$.*

Proof. Available in the online appendix and the extended version [Binder et al. 2024b]. □

6 DE/REFUNCTIONALIZATION

In our system, de- and refunctionalization can transform any data type into a codata type and vice versa. The key insight behind these transformations is that any data or codata type in the program can be represented in matrix form as shown in Figure 11. A data type is fully determined by specifying an expression for each constructor-definition pair, while a codata type is fully determined by giving an expression for each destructor-codefinition pair. Using these matrix representations, the process of de- and refunctionalization simplifies to matrix transposition. With this intuition in mind, we will now formally define de- and refunctionalization and state our main propositions.

Ctor \ Def	$(z_1 : \mathcal{T}\rho_1).d_1\Xi_1 : t_1$	$\cdots$	$(z_m : \mathcal{T}\rho_m).d_m\Xi_m : t_m$
$C_1\Xi_1 : \mathcal{T}\rho_1$	$e_{1,1}$	$\cdots$	$e_{1,m}$
$\vdots$	$\vdots$		$\vdots$
$C_n\Xi_n : \mathcal{T}\rho_n$	$e_{n,1}$	$\cdots$	$e_{n,m}$

(a) The data matrix.

Dtor \ Codef	$C_1\Xi_1 : \mathcal{T}\rho_1$	$\cdots$	$C_n\Xi_n : \mathcal{T}\rho_n$
$(z_1 : \mathcal{T}\rho_1).d_1\Xi_1 : t_1$	$e_{1,1}$	$\cdots$	$e_{n,1}$
$\vdots$	$\vdots$		$\vdots$
$(z_m : \mathcal{T}\rho_m).d_m\Xi_m : t_m$	$e_{1,m}$	$\cdots$	$e_{n,m}$

(b) The codata matrix.

Fig. 11. Data and codata matrix.

Definition 6.1 (Defunctionalization). We write $\mathcal{D}_\mathcal{T}(\Theta)$ to denote the defunctionalization of $\mathcal{T}$ in Θ. The precondition for applying $\mathcal{D}_\mathcal{T}(\Theta)$ is that **codata** $\mathcal{T}\ \Psi\ \{\ \ldots\ \} \in \Theta$. Defunctionalization transforms the codata type into a data type. For each destructor d and each codefinition C in Θ:

$$\textbf{codata}\ \mathcal{T}\ \Psi\ \{\ (z : \mathcal{T}\rho_2).\ d\ \Xi_2 : t, \ldots\ \}, \textbf{codef}\ C\ \Xi_1 : \mathcal{T}\ \rho_1\ \{\ d\ \Xi_2 \mapsto e, \ldots\ \}$$

the program $\mathcal{D}_\mathcal{T}(\Theta)$ contains a corresponding constructor C and a definition d:

$$\textbf{data}\ \mathcal{T}\ \Psi\ \{\ C\ \Xi_1 : \mathcal{T}\ \rho_1, \ldots\ \}, \textbf{def}\ (z : \mathcal{T}\ \rho_2).d\ \Xi_2 : t\ \{\ C\ \Xi_1 \mapsto e, \ldots\ \}$$

Definition 6.2 (Refunctionalization). We write $\mathcal{R}_\mathcal{T}(\Theta)$ to denote the refunctionalization of $\mathcal{T}$ in Θ. The precondition for applying $\mathcal{R}_\mathcal{T}(\Theta)$ is that **data** $\mathcal{T}\ \Psi\ \{\ \ldots\ \} \in \Theta$. Refunctionalization transforms the data type into a codata type. For each constructor C and each definition d in Θ:

$$\textbf{data}\ \mathcal{T}\ \Psi\ \{\ C\ \Xi_1 : \mathcal{T}\ \rho_1, \ldots\ \}, \textbf{def}\ (z : \mathcal{T}\ \rho_2).d\ \Xi_2 : t\ \{\ C\ \Xi_1 \mapsto e, \ldots\ \}$$

the program $\mathcal{R}_\mathcal{T}(\Theta)$ contains a corresponding codefinition C and a destructor d:

$$\textbf{codata}\ \mathcal{T}\ \Psi\ \{\ (z : \mathcal{T}\rho_2).\ d\ \Xi_2 : t, \ldots\ \}, \textbf{codef}\ C\ \Xi_1 : \mathcal{T}\ \rho_1\ \{\ d\ \Xi_2 \mapsto e, \ldots\ \}$$

We write $\mathcal{X}_\mathcal{T}(\cdot)$ for both $\mathcal{D}_\mathcal{T}(\cdot)$ and $\mathcal{R}_\mathcal{T}(\cdot)$, when their difference doesn't matter. Using these definitions, we can now state our main propositions. Notice that de- and refunctionalization do not affect the program on the expression level. This is because we reuse the syntax for *producers* for

both constructor and codefinition calls and the syntax for *consumers* for both destructor and definition calls (see Figure 9). Therefore, in the proposition statements below, de-/refunctionalization is only applied to the program Θ.

Theorem 6.3 (De/Refunctionalization preserves typing and judgmental equality). *The following implications hold:*

- $\Gamma \vdash_\Theta e : t \quad \Longrightarrow \quad \Gamma \vdash_{\mathcal{X}_\mathcal{T}(\Theta)} e : t$
- $\Gamma \vdash_\Theta e_1 \equiv e_2 : t \quad \Longrightarrow \quad \Gamma \vdash_{\mathcal{X}_\mathcal{T}(\Theta)} e_1 \equiv e_2 : t$
- $\Gamma \vdash_\Theta \sigma : \Xi \quad \Longrightarrow \quad \Gamma \vdash_{\mathcal{X}_\mathcal{T}(\Theta)} \sigma : \Xi$
- $\Gamma \vdash_\Theta \sigma_1 \equiv \sigma_2 : \Xi \quad \Longrightarrow \quad \Gamma \vdash_{\mathcal{X}_\mathcal{T}(\Theta)} \sigma_1 \equiv \sigma_2 : \Xi$
- $\vdash_\Theta \Gamma\ \mathrm{ctx} \quad \Longrightarrow \quad \vdash_{\mathcal{X}_\mathcal{T}(\Theta)} \Gamma\ \mathrm{ctx}$
- $\Gamma \vdash_\Theta \Xi\ \mathrm{tel} \quad \Longrightarrow \quad \Gamma \vdash_{\mathcal{X}_\mathcal{T}(\Theta)} \Xi\ \mathrm{tel}$

Proof. Proof outline available in the online appendix and the extended version [Binder et al. 2024b]. □

Theorem 6.4 (De/Refunctionalization preserves well-formedness of programs). *If* $\vdash_\Theta \Theta\ \mathit{Ok}$, *then* $\vdash_\Theta \mathcal{X}_\mathcal{T}(\Theta)\ \mathit{Ok}$

Proof. Proof outline available in the online appendix and the extended version [Binder et al. 2024b]. □

7 FUTURE WORK

In this paper, we described a dependently typed programming language based on data and codata. How to extend this programming language to a proof assistant is one of the problems that we want to address in the future. In the following sections, we describe the problems that have to be solved to make our system consistent, in a way that is compatible with the transformations we described.

7.1 Specifying Termination and Productivity

The system we presented does not have any form of termination or productivity checking. We could, of course, use any of the existing off-the-shelf solutions for checking termination and productivity. The problem with that approach is that, in general, a program that typechecks and is verified to only have terminating recursive definitions and productive corecursive definitions might not be verifiably total after de/-refunctionalization. We illustrate this with the following example:

```
data Nat { S(x: Nat), Z }
def Nat.plus(n: Nat): Nat {
  Z => n,
  S(x') => S(x'.plus(n)) }
def Nat.mul(n: Nat): Nat {
  Z => Z,
  S(m) => n.plus(m.mul(n)) }
```

```
codata Nat { plus(n: Nat): Nat, mul(n: Nat): Nat }
codef S(x: Nat): Nat {
  plus(n) => S(x.plus(n)),
  mul(n) => n.plus(x.mul(n)) }
codef Z: Nat {
  plus(n) => n,
  mul(n) => Z }
```

We could check termination for the program on the left in the usual way. We check the definition of `plus` first and verify that it is only called on the structurally smaller argument `m`. We then add `plus` to the context of functions which are checked to be total. We then check the definition of `mul`, having the function `plus` as a total function in the context. We see again that `mul` is called on the structurally smaller argument `m`.

Refunctionalizing the program on the left results in the program on the right. In this program, we have to check the productivity of the definitions of `Z` and `S`. If we want de/refunctionalization to be a transformation that maps valid programs to valid programs, then the evidence for the productivity of `Z` and `S` has to be composed of the evidence for the termination of `plus` and `mul`. But it is not at all clear how this can be formally specified at the moment.

7.2 Universe Hierarchy

Another reason why our system is inconsistent is that we use one impredicative universe with the Type-in-Type axiom. This axiom is known to make the theory inconsistent [Hurkens 1995]; on the other hand, it vastly simplifies the presentation and implementation of the theory if we don't have to care about universe levels. We want to investigate how de- and refunctionalization interact with the assignment of type universes to types. This was also identified as a problem by Huang and Yallop [2023].

7.3 The Variance Problem

Most proof assistants enforce *strict positivity* in the definition of data types. The strict positivity restriction says that recursive occurrences of the type that is defined are only allowed at strictly positive positions, and is required to avoid Curry's paradox.

The only source of contravariance in most systems is the function type, where arguments are contravariant. In a system with user-defined codata types many different types are the source of contravariance, but this is quite simple to specify. The problem is that many useful types from object-oriented programming require both positive and negative occurrences of the type being defined. For example, consider the following type[7]:

```
codata NatSet { member(x: Nat): Bool, union(x: NatSet): NatSet }
```

In this example, the `NatSet` type occurs both positively and negatively in the `union` destructor. But this definition is a sensible one; it is not an obscure definition at all. So if we want to enable the user to work with such definitions we have to replace the strict positivity check by something more refined. One avenue that we want to explore is guarded type theory (e.g. [Clouston et al. 2017]). Guarded logic was introduced by [Nakano 2000] precisely in order to fix problems with binary methods in object-oriented programming, and was later developed by other authors into guarded type theories.

7.4 Strong Behavioural Equality

There is no universally satisfactory definition of equality as the appropriate definition depends on the object being modeled. For many data types, syntactic equality is sensible. For instance, two natural numbers $n_0, n_1 : \mathbb{N}$ are considered judgmentally equal if they are built from the same constructors, i.e. $\mathtt{Z} \equiv \mathtt{Z}$ and $n_0 \equiv n_1 \implies \mathtt{S}(n_0) \equiv \mathtt{S}(n_1)$. However, the situation is very different as soon as we consider functions $f, g : \mathbb{N} \to \mathbb{N}$. Syntactic equality of the function definitions does not seem appropriate, because multiple definitions can define the *same* function. A more reasonable approach is to regard two functions as equal if they behave identically on all inputs. This principle is known as functional extensionality:

$$\mathtt{fun\_ext} : \forall f\ g, (\forall x, \mathtt{Eq}(\mathbb{N} \to \mathbb{N}, f\ x, g\ x)) \implies \mathtt{Eq}(\mathbb{N} \to \mathbb{N}, f, g)$$

Functional extensionality is the prototypical example of behavioral equality. It is a special case of *bisimilarity*: We consider any two objects equal if they behave identically with regard to their observations. For codata types, we often desire behavioral equalities such as bisimilarity.

In most proof assistants, one can pose those propositional behavioral equalities as axioms. But this approach does not work in our system. This is because the functional extensionality axiom is inconsistent for the data representation `Fun` of functions $\mathbb{N} \to \mathbb{N}$, which can be seen in the following example:

[7] This example was pointed out in the answer by Neel Krishnaswami to the following question on the proof assistants stack exchange: https://proofassistants.stackexchange.com/questions/372/bringing-oop-features-into-proof-assistants.

```
data Fun { Id1, Id2 }
def Fun.ap(x: Nat): Nat { Id1 => x, Id2 => x }
codef apply_id1_eq_id2: Π(Nat, \x. Eq(Fun, Id1.ap(x), Id2.ap(x))) {
    pi_elim(_, _, x) => Refl(Fun, x)
}
fun_ext(Id1, Id2, apply_id1_eq_id2): Eq(Fun, Id1, Id2)
```

Here we can derive the propositional equality `Eq(Fun, Id1, Id2)`, which is a contradiction to the provable proposition that `Id1` is not propositionally equal to `Id2`. Hence, this counterexample shows that we cannot formulate behavioral equalities as axioms in our system.

However, stating behavioral equalities as axioms is often considered unsatisfactory. Axioms break canonicity, i.e., the property that any closed term can be reduced to a canonical value. Further, bisimilarity is always relative to a given set of observations. To see this, consider the functional extensionality axiom from above. It assumes that the function type has a single elimination form, namely function application. Identifying all types that behave identically on application is, without further conditions, only reasonable if application is the only observation we can make. Hence, we have the principal problem that it is unclear how to specify behavioral equalities given that we are extensible in the observations.

Luckily, there are better approaches for working with behavioral equality that are expected to be compatible with our system. For instance, we can resort to the typical Setoid approach of defining a type together with an equality relation:

```
codata Setoid {
    type: Type,
    (self: Setoid).equality: Fun(self.type, Fun(self.type, Type)) }
```

Unfortunately, the Setoid approach is known for bad usability. To achieve a more ergonomic solution, we need a system that allows us to define a type in conjunction with its equalities. In particular, it must be possible for distinctly named constructors to be equal. Future work in this direction could look into extending de- and refunctionalization to observational type theory [Altenkirch and McBride 2006] or a system with higher inductive types.

8 RELATED WORK

Codata types. Codata types were first introduced by Hagino [Hagino 1987, 1989]. The original interpretation of codata types stems from coalgebras in category theory. An overview of the history of codata types as coalgebras is given by Setzer [Setzer 2012]. We have discussed the relation between codata types and OOP in Section 1.1.

The expression problem. The expression problem poses the challenge for statically typed languages to create a type which can be extended by both new producers and new consumers. Based on earlier observations, Wadler [1998] formulated the problem and gave it its current name. The expression problem for proofs is recognized as an important challenge in the verification community, but there are fewer proposed and implemented solutions than in the programming world. One popular solution in the programming world is Swierstra [2008]'s "Data Types à la carte" approach. In that approach, a type is defined as the fixpoint of a coproduct of functors which can be extended by new functors in a modular way. Most proposed solutions for dependent types are based on Swierstra [2008]'s approach and extend them to dependent types. Delaware et al. [2013a,b]; Delaware [2013] as well as Keuchel and Schrijvers [2013] implemented this approach for the Coq proof assistant and Schwaab and Siek [2013] implemented it for Agda. A system for writing modular proofs in the Isabelle proof assistant has been described by Molitor [2015]. The most recent adaptation of the idea is by Forster and Stark [2020], who give an excellent presentation of this line of work in their related work section.

In this article, our focus was not to propose a *solution* to the expression problem for proofs, since the defunctionalization and refunctionalization algorithms are whole-program transformations. Instead, our approach *distills the essence* of the expression problem for dependent types: Neither the functional nor the object-oriented decomposition solves the expression problem, since data types cannot be easily extended by new constructors, and codata types cannot be easily extended by new destructors.

Dependently-typed object-oriented programming. In Section 2 we presented our perspective on dependently-typed object-oriented programming. But we are not the first to think about this design space. Jacobs [1995] proposes using coalgebras to express object-oriented classes with coalgebraic specifications. His concept is based on three main components: objects, class implementations, and class specifications. The latter are used to specify a set of methods on an abstract state space as well as a set of assertions that define the behavior of these methods. Such a specification can then be implemented by a class. A class gives a carrier set as a concrete interpretation for the state space and a coalgebra that implements the specified methods. An object is then just an element of the carrier set. In our system, we can express specifications similar to Jacobs' proposal using self-parameters on destructors (see Section 2.2). Rather than having separate notions of specifications, classes, and objects, our system has a singular notion of codata types. Jacobs separates these notions to construct a model in which objects are indistinguishable if they are bisimilar according to their specification. In contrast, in our system, we have a full syntactic duality between data and codata types through de- and refunctionalization. Hence, we need to decouple codata types from the semantics that are usually associated with them, including behavioral equality such as bisimilarity. Setzer [2006] conceived of dependently-typed object-oriented programming by specifying interfaces and having *interactive programs* as objects implementing these interfaces. The interfaces contain a *command type*, which represents the method signatures of an interface. Interactive programs are programs that react to incoming method calls by producing a return value and a new object.

Dependent type theories with definable Π*-type and* Σ*-type.* In Section 2.3 we demonstrated that the programmer can define both the Π-type and the Σ-type in our system, whereas in most proof assistants only the Σ-type can be defined. This is a generalization of the observation that programmers can't define the function type in most functional programming languages, but that the function type can be defined in object-oriented languages [Setzer 2003]. Apart from this paper, the only other dependent type theory that doesn't presuppose a built-in Π-type is by Basold [2018]; Basold and Geuvers [2016]. Like us, they give an explicit definition of the Π-type in their system. Their definition, however, is slightly different from ours, since they have a more expressive core system. In their system, parameterized type constructors and type variables don't have to be fully applied. Partially applied type constructors have a special type $\Gamma \twoheadrightarrow *$, and they specify a sort of simply-typed lambda calculus which governs the rules for abstracting over, and partially applying type constructors to arguments. As a result, their definition of the Π and Σ-type is a bit simpler: We have to use a previously-defined non-dependent function type $A \to \text{Type}$ to represent the type family that the Σ and Π-types are indexed over, while they use a partially applied type variable $X : A \twoheadrightarrow *$. Our system is also not consistent; theirs is, and they prove both subject reduction and strong normalization.

Dependent pattern matching. The traditional primitive elimination forms in dependent type theories are *eliminators*. The eliminator for natural numbers, for example, has the type $\forall(\mathrm{P} : \mathbb{N} \to \text{Type}), \mathrm{P}\ 0 \to (\forall \mathrm{n} : \mathbb{N}, \mathrm{P}\ \mathrm{n} \to \mathrm{P}\ (\mathrm{S}\ \mathrm{n})) \to \forall \mathrm{n} : \mathbb{N}, \mathrm{P}\ \mathrm{n}$. They are suitable for studying the metatheory

of dependent types, but programming with them isn't very ergonomic. A more convenient alternative to eliminators is dependent pattern matching, a generalization of ordinary pattern matching to dependent types, which was first proposed by Coquand [1992]. While ordinary pattern matching can be compiled to eliminators and is therefore nothing more than syntactic sugar, the compilation of dependent pattern matches additionally requires Streicher [1993]'s axiom K. Hofmann and Streicher [1994] proved that this axiom does not follow from the standard elimination rules for the identity type. Since the K axiom is sometimes undesirable—it is incompatible with other principles such as univalence—a variant of dependent pattern matching which does not rely on axiom K was developed by Cockx et al. [2014]. We use a variant of dependent pattern and copattern matching which requires axiom K if we want to compile it to eliminators, but we could get rid of this dependency by applying the three restrictions presented in Cockx' thesis [Cockx 2017, p.55].

Copattern matching. Copattern matching as a dual concept to pattern matching was first proposed by Abel et al. [2013]. Their work was motivated by the deficiencies of previous approaches which used constructors to represent infinite objects. For instance, the coinductive types originally introduced in Coq broke subject reduction, as noted by Giménez [1996] and Oury [2008]. Even simple infinite objects such as streams cannot be represented using constructors and pattern matching in a sensible way. This follows from the observation of Berger and Setzer [2018] that there exists no decidable equality for streams which admits a one-step expansion of a stream s to a stream $(\text{cons}\ n\ s')$.

Inconsistent dependent type theories. The type theory presented in this paper is inconsistent, i.e. every type is inhabited by some term, a property it shares with most programming languages but not with proof assistants. However, the inconsistency of the theory does not imply that the properties expressed by the dependent types are meaningless. We can compare the situation to the programming language Haskell, where it is already possible to write dependent programs by using several language extensions and programming tricks [Eisenberg and Weirich 2012; Lindley and McBride 2013]. Instead of relying on these tricks, a more ergonomic and complete design of dependent types in Haskell has been the subject of various articles [Weirich et al. 2019, 2017] and PhD theses [Eisenberg 2016; Gundry 2013]. Their main insight also applies to our system: the central property of an inconsistent dependent type theory is type soundness [Wright and Felleisen 1994]. For example, every term of type `Vec(5)` can only evaluate to a vector containing five elements or diverge; it cannot evaluate to a vector of six elements. But they also show that inconsistency has downsides, especially for optimization: In a consistent theory every term of type $\texttt{Eq}(s, t)$ must evaluate to the term `refl`, and can therefore be erased during compilation. In an inconsistent theory, we cannot erase the equality witness, since we could otherwise write a terminating unsafe coercion between arbitrary types, which would violate type soundness.

Defunctionalization and refunctionalization. The related work on defunctionalization and refunctionalization can be partitioned into two groups: The first group only considers defunctionalization and refunctionalization for the function type, while the second group generalizes them to transformations between arbitrary data and codata types. De/Refunctionalization of the function type has a long history, which starts with the seminal paper by Reynolds [1972] and the later work of Danvy and Millikin [2009]; Danvy and Nielsen [2001]. That the defunctionalization of polymorphic functions requires GADTs was first observed by Pottier and Gauthier [2006]. In a recent paper, Huang and Yallop [2023] describe the defunctionalization of dependent functions, and especially how to correctly deal with type universes and positivity restrictions, but don't consider the general case of indexed data and codata types. On the contrary, they do not use data types at all

and instead introduce the construct of first-class function labels which enables them to avoid problems arising from the expressivity of data type definitions like recursive types. The generalization of defunctionalization from functions to arbitrary codata types was first described by Rendel et al. [2015] for a simply typed system without local lambda abstractions or local pattern matches. That the generalization to polymorphic data and codata types then also requires GAcoDTs has been described by Ostermann and Jabs [2018]. How to treat local pattern and copattern matches in such a way as to preserve the invertibility of defunctionalization and refunctionalization has been described by Binder et al. [2019]. Recently, Zhang et al. [2022] implemented defunctionalization and refunctionalization for the programming language Scala, and used these transformations for some larger case studies. In this article, we describe the generalization to indexed data and codata types, but in distinction to Huang and Yallop [2023] we circumvent the problems of type universes and positivity restrictions by working in an inconsistent type theory.

9 CONCLUSION

Most dependently typed programming languages don't support programming with codata as well as programming with data. The main reason some proof assistants support codata types at all was that some support was necessary for the convenient formalization of theorems about infinite and coalgebraic objects. But codata types are useful for more than just representing infinite objects like streams; they represent an orthogonal way to structure programs and proofs, with different extensibility properties and reasoning principles. In this paper we have presented a vision of how programming can look in a dependently typed language in which the data and codata sides are completely symmetric and treated with equal care. By implementing this language and testing it on a case study we have demonstrated that this style of purely functional, dependently typed object-oriented programming does work. We think that this way of systematic language design, in place of ad-hoc extensions, provides a good case study on how the design of dependently typed languages and proof assistants should be approached.

DATA-AVAILABILITY STATEMENT

This article is accompanied by an online IDE available at polarity-lang.github.io/oopsla24 where the examples discussed in this paper can be selected and loaded. This online IDE consists of a static website hosted on GitHub pages, with all the code running in the browser on the client side. Should the hosted website despite our best efforts no longer be available, then it is possible to recreate it locally using the archived version available at Zenodo [Binder et al. 2024a].

REFERENCES

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2480359.2429075

Thorsten Altenkirch and Conor McBride. 2006. Towards Observational Type Theory. http://www.strictlypositive.org/ott.pdf

Henning Basold. 2018. *Mixed Inductive-Coinductive Reasoning: Types, Programs and Logic.* Ph. D. Dissertation. Radboud University. https://hdl.handle.net/2066/190323

Henning Basold and Herman Geuvers. 2016. Type Theory Based on Dependent Inductive and Coinductive Types. In *Proceedings of the Symposium on Logic in Computer Science* (New York). Association for Computing Machinery, New York, NY, USA, 327–336. https://doi.org/10.1145/2933575.2934514

Ulrich Berger and Anton Setzer. 2018. Undecidability of Equality for Codata Types. In *Coalgebraic Methods in Computer Science*, Corina Cîrstea (Ed.). Springer, 34–55. https://doi.org/10.1007/978-3-030-00389-0_4

David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. 2019. Decomposition Diversity with Symmetric Data and Codata. *Proc. ACM Program. Lang.* 4, POPL, Article 30 (2019), 28 pages. https://doi.org/10.1145/3371098

David Binder, Ingo Skupin, Tim Süberkrüb, and Klaus Ostermann. 2024a. *Deriving Dependently-Typed OOP from First Principles*. https://doi.org/10.5281/zenodo.10779424 Archived version of the submitted artefact.

David Binder, Ingo Skupin, Tim Süberkrüb, and Klaus Ostermann. 2024b. Deriving Dependently-Typed OOP from First Principles – Extended Version with Additional Appendices. (2024). https://doi.org/10.48550/arXiv.2403.06707

Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2017. The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types. *Logical Methods in Computer Science* 12 (April 2017). Issue 3. https://doi.org/10.2168/LMCS-12(3:7)2016

Jesper Cockx. 2017. *Dependent Pattern Matching and Proof-Relevant Unification*. Ph. D. Dissertation. KU Leuven.

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014. Pattern Matching without K. In *International Conference on Functional Programming*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/2628136.2628139

William R. Cook. 1990. Object-Oriented Programming versus Abstract Data Types. In *Proceedings of the REX Workshop / School on the Foundations of Object-Oriented Languages*. Springer, 151–178. https://doi.org/10.1007/BFb0019443

William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications: Onward! Essays* (Orlando). Association for Computing Machinery, New York, NY, USA, 557–572. https://doi.org/10.1145/1640089.1640133

Thierry Coquand. 1992. Pattern Matching With Dependent Types. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs* (Bastad, Sweden), Bengt Nordström, Kent Pettersson, and Gordon Plotkin (Eds.). 66–79.

Olivier Danvy and Kevin Millikin. 2009. Refunctionalization at Work. *Science of Computer Programming* 74, 8 (2009), 534–549. https://doi.org/10.1016/j.scico.2007.10.007

Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *Proceedings of the Conference on Principles and Practice of Declarative Programming* (Florence). 162–174. https://doi.org/10.1145/773184.773202

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013a. Meta-Theory à La Carte. In *Symposium on Principles of Programming Languages* (Rome) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/2429069.2429094

Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. 2013b. Modular Monadic Meta-Theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 319–330. https://doi.org/10.1145/2500365.2500587

Benjamin James Delaware. 2013. *Feature Modularity in Mechanized Reasoning*. Ph. D. Dissertation. The University of Texas at Austin.

Paul Downen, Zachary Sullivan, Zena M. Ariola, and Simon Peyton Jones. 2019. Codata in Action. In *European Symposium on Programming (ESOP '19)*. Springer, 119–146. https://doi.org/10.1007/978-3-030-17184-1_5

Richard Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice*. Ph. D. Dissertation. University of Pennsylvania.

Richard A. Eisenberg, Guillaume Duboc, Stephanie Weirich, and Daniel Lee. 2021. An Existential Crisis Resolved: Type Inference for First-Class Existential Types. *Proc. ACM Program. Lang.* 5, ICFP, Article 64 (aug 2021), 29 pages. https://doi.org/10.1145/3473569

Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Proceedings of the Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 117–130. https://doi.org/10.1145/2364506.2364522

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science* 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

Yannick Forster and Kathrin Stark. 2020. Coq à La Carte: A Practical Approach to Modular Syntax with Binders. In *Proceedings of the Conference on Certified Programs and Proofs* (New Orleans) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 186–200. https://doi.org/10.1145/3372885.3373817

Peng Fu and Aaron Stump. 2014. Self Types for Dependently Typed Lambda Encodings. In *International Conference on Rewriting Techniques and Applications*, Gilles Dowek (Ed.). Springer, 224–239. https://doi.org/10.1007/978-3-319-08918-8_16

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co., Boston.

Richard Garner. 2009. On the Strength of Dependent Products in the Type Theory of Martin-Löf. *Annals of Pure and Applied Logic* 160, 1 (2009), 1–12. https://doi.org/10.1016/j.apal.2008.12.003

Herman Geuvers. 2001. Induction Is Not Derivable in Second Order Dependent Type Theory. In *Typed Lambda Calculi and Applications*, Samson Abramsky (Ed.). Springer, Berlin, Heidelberg, 166–181. https://doi.org/3-540-45413-6_16

Herman Geuvers. 2014. The Church-Scott Representation of Inductive and Coinductive Data. https://www.cs.vu.nl/~femke/courses/ep/slides/herman-data-types.pdf Presented at the TYPES 2014 meeting.

Eduardo Giménez. 1996. *Un Calcul de Constructions Infinies et son application a la vérification de systemes communicants*. Ph. D. Dissertation. Lyon, École normale supérieure (sciences).

Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thése de Doctorat d'Etat. Université de Paris VII.

Brian Goetz et al. 2014. *JSR 335: Lambda Expressions for the Java Programming Language*. https://jcp.org/en/jsr/detail?id=335

Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types*. Ph. D. Dissertation. University of Strathclyde.

Tatsuya Hagino. 1987. *A Categorical Programming Language*. Ph. D. Dissertation. University of Edinburgh. https://doi.org/10.48550/arXiv.2010.05167

Tatsuya Hagino. 1989. Codatatypes in ML. *Journal of Symbolic Computation* 8, 6 (1989), 629–650. https://doi.org/10.1016/S0747-7171(89)80065-3

Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Extensional Constructs in Intensional Type Theory*. Springer, London, 13–54. https://doi.org/10.1007/978-1-4471-0963-1_2

Martin Hofmann and Thomas Streicher. 1994. The Groupoid Model Refutes Uniqueness of Identity Proofs. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 208–212. https://doi.org/10.1109/LICS.1994.316071

William Alvin Howard. 1980. The Formulae-as-Types Notion of Construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan (Eds.). Academic Press.

Yulong Huang and Jeremy Yallop. 2023. Defunctionalization with Dependent Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 127 (jun 2023), 23 pages. https://doi.org/10.1145/3591241

Antonius J. C. Hurkens. 1995. A Simplification of Girard's Paradox. In *Proceedings of the Conference on Typed Lambda Calculi and Applications*. Springer, London, 266–278. http://dx.doi.org/10.1007/BFb0014058

Bart Jacobs. 1995. Objects and Classes, Coalgebraically. In *Object Orientation with Parallelism and Persistence*, Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek (Eds.). Springer, 83–103. https://doi.org/10.1007/978-1-4613-1437-0_5

Steven Keuchel and Tom Schrijvers. 2013. Generic Datatypes à La Carte. In *Workshop on Generic Programming* (Boston) *(WGP '13)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2502488.2502491

Pieter Koopman, Rinus Plasmeijer, and Jan Martin Jansen. 2014. Church Encoding of Data Types Considered Harmful for Implementations: Functional Pearl. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*. Association for Computing Machinery, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/2746325.2746330

Sam Lindley and Conor McBride. 2013. Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming. In *Proceedings of the Haskell Symposium* (Boston) *(Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/2503778.2503786

Richard Molitor. 2015. *Open Inductive Predicates*. Master's thesis. Karlsruher Institut für Technologie (KIT).

Hiroshi Nakano. 2000. A Modality for Recursion. In *Proceedings of the Symposium on Logic in Computer Science*. 255–266. https://doi.org/10.1109/LICS.2000.855774

Klaus Ostermann and Julian Jabs. 2018. Dualizing Generalized Algebraic Data Types by Matrix Transposition. In *European Symposium on Programming*. Springer, 60–85. https://doi.org/10.1007/978-3-319-89884-1_3

Nicolas Oury. 2008. Coinductive Types and Type Preservation. *Message on the coq-club mailing list* (2008). https://sympa.inria.fr/sympa/arc/coq-club/2008-06/msg00022.html

François Pottier and Nadji Gauthier. 2006. Polymorphic Typed Defunctionalization and Concretization. *Higher-Order and Symbolic Computation* 19, 1 (3 2006), 125–162. https://doi.org/10.1007/s10990-006-8611-7

Tillmann Rendel, Julia Trieflinger, and Klaus Ostermann. 2015. Automatic Refunctionalization to a Language with Copattern Matching: With Applications to the Expression Problem. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 269–279. https://doi.org/10.1145/2784731.2784763

John Charles Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *ACMConf* (Boston). Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

Christopher Schwaab and Jeremy G. Siek. 2013. Modular Type-Safety Proofs in Agda. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification* (Rome, Italy) *(PLPV '13)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/2428116.2428120

Anton Setzer. 2003. Java as a Functional Programming Language. In *Types for Proofs and Programs*, Herman Geuvers and Freek Wiedijk (Eds.). Springer, Berlin, Heidelberg, 279–298. https://doi.org/10.1007/3-540-39185-1_16

Anton Setzer. 2006. Object-Oriented Programming in Dependent Type Theory. *Trends in Functional Programming* 7 (2006), 91–108.

Anton Setzer. 2012. Coalgebras as Types Determined by Their Elimination Rules. In *Epistemology versus Ontology. Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf*, Peter Dybjer, Sten Lindström, Erik Palmgren, and Göran Sundholm (Eds.). Logic, Epistemology, and the Unity of Science, Vol. 27. Springer, Dordrecht, 351–369. https://doi.org/10.1007/978-94-007-4435-6_16

Thomas Streicher. 1993. Investigations into Intensional Type Theory. Habilitationsschrift, Ludwig-Maximilians-Universität München.

Wouter Swierstra. 2008. Data Types à la Carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Neil Tennant. 1982. Proof and Paradox. *Dialectica* 36, 2-3 (1982), 265–296. https://doi.org/10.1111/j.1746-8361.1982.tb00820.x

David Thibodeau, Andrew Cave, and Brigitte Pientka. 2016. Indexed Codata Types. In *Proceedings of the International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 351–363. https://doi.org/10.1145/2951913.2951929

Philip Wadler. 1998. The Expression Problem. (11 1998). https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt Note to Java Genericity mailing list.

Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A Role for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 101 (jul 2019), 29 pages. https://doi.org/10.1145/3341705

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proceedings of the ACM on Programming Languages* 1, ICFP (8 2017). https://doi.org/10.1145/3110275

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (11 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Weixin Zhang, Cristina David, and Meng Wang. 2022. Decomposition Without Regret. *arXiv preprint arXiv:2204.10411* (2022). https://doi.org/10.48550/ARXIV.2204.10411

Received 21-OCT-2023; accepted 2024-02-24