Symmetric Dependent Data and Codata Types

Programming Languages Seminar, TU Delft

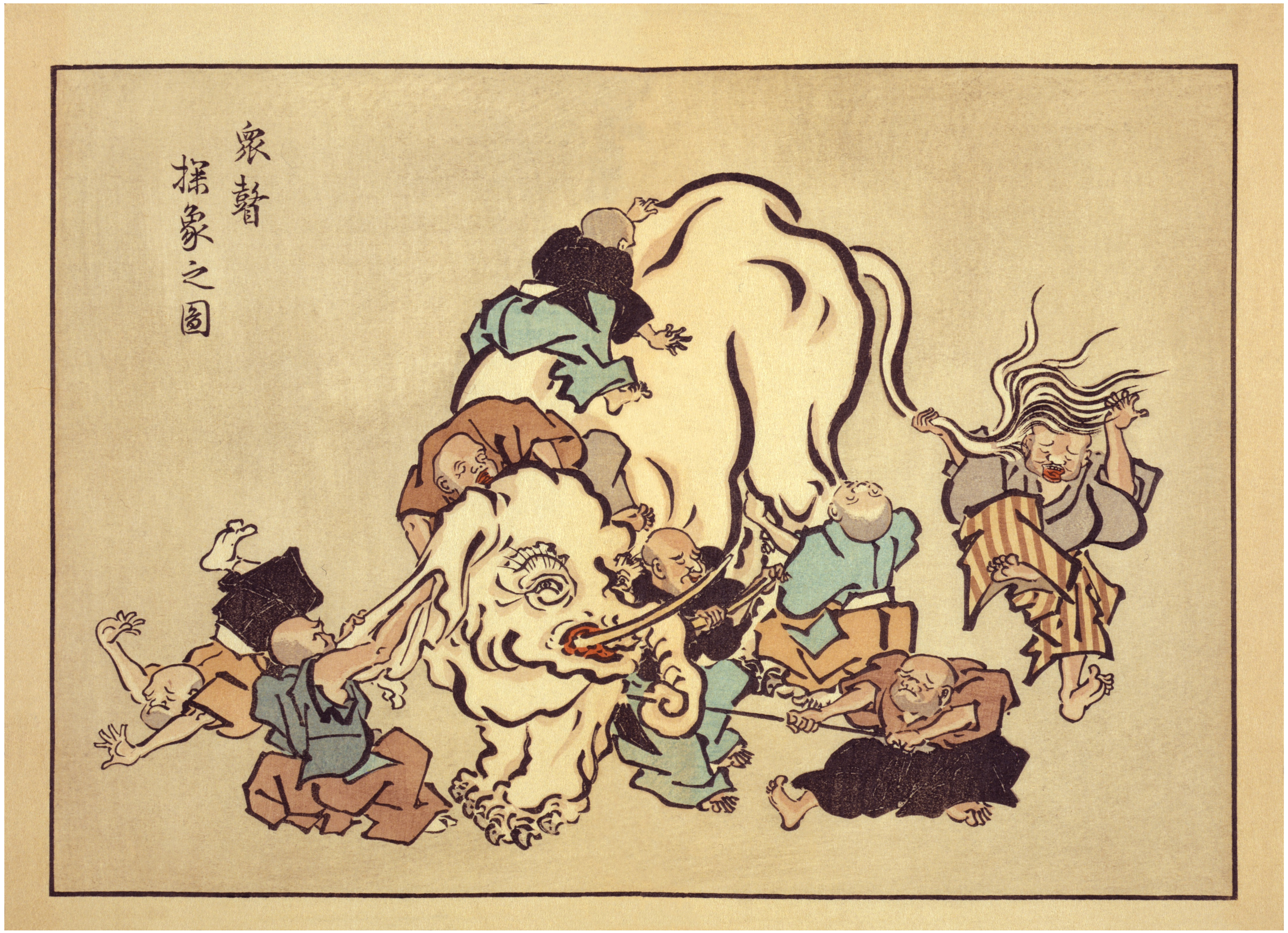
Slides available: binderdavid.github.io -> Talks

David Binder, University of Tübingen, 2024

What codata types can you think of?

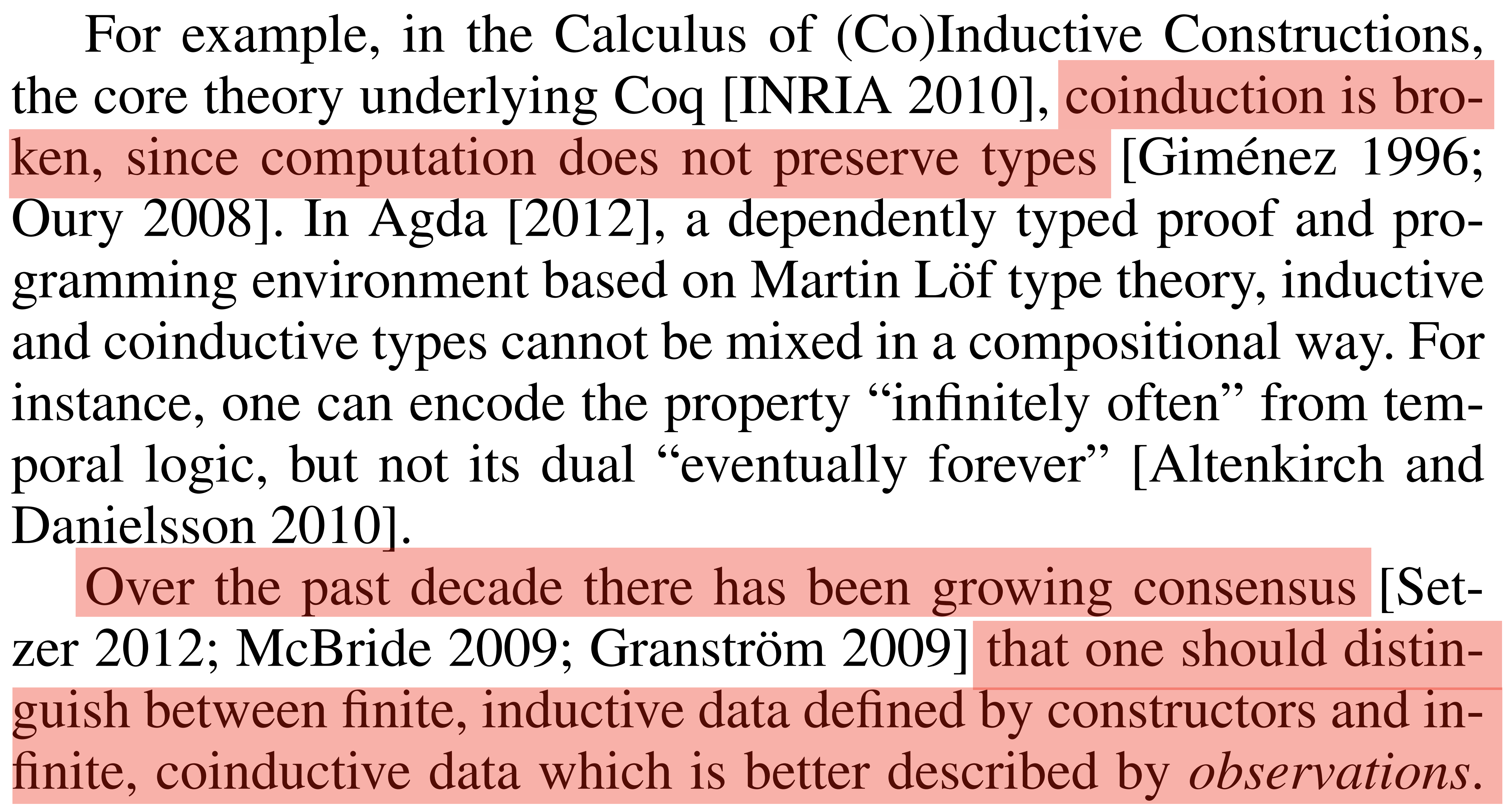
This Talk:
Codata Types are Types Defined by Observations

Codata Types: Five Sketches


衆瞽
探象之図

Sketch 1

Codata for Infinite Data



For example, in the Calculus of (Co)Inductive Constructions, the core theory underlying Coq [INRIA 2010], coinduction is broken, since computation does not preserve types [Giménez 1996; Oury 2008]. In Agda [2012], a dependently typed proof and programming environment based on Martin Löf type theory, inductive and coinductive types cannot be mixed in a compositional way. For instance, one can encode the property "infinitely often" from temporal logic, but not its dual "eventually forever" [Altenkirch and Danielsson 2010].

Over the past decade there has been growing consensus [Setzer 2012; McBride 2009; Granström 2009] that one should distinguish between finite, inductive data defined by constructors and infinite, coinductive data which is better described by *observations*.

Copatterns: Programming Infinite Structures by Observations (Abel, Pientka, Thibodeau, Setzer)

Codata for Infinite Data

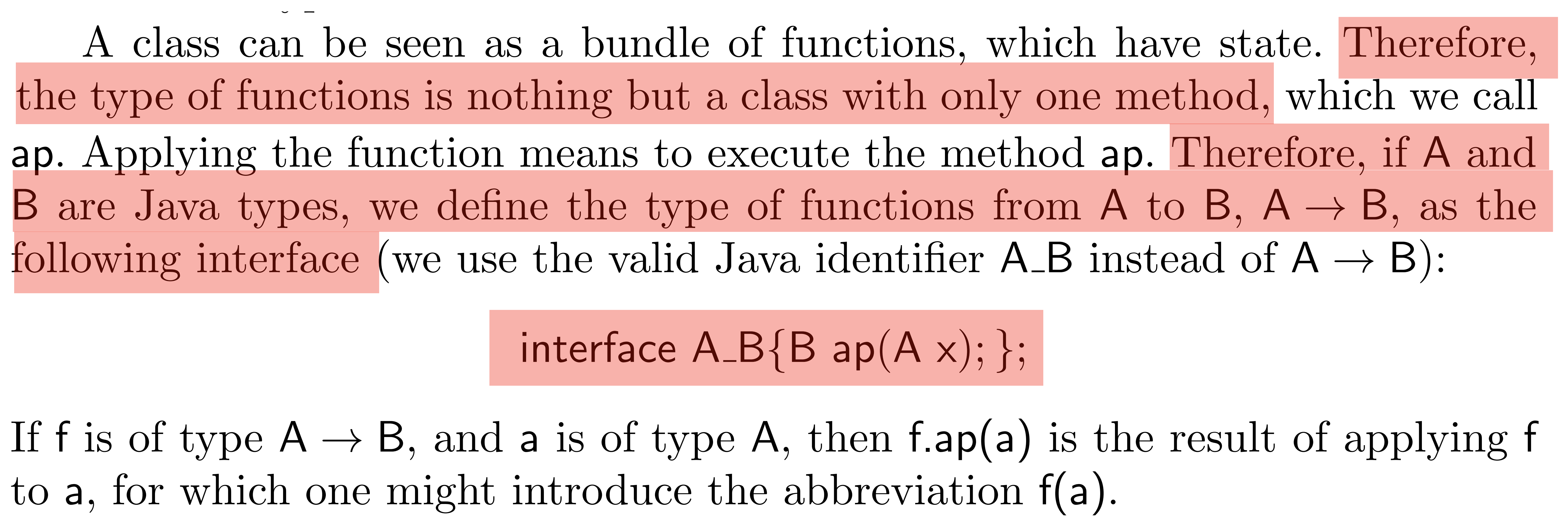
The codata type Stream

- CoInductive Stream(A : Set) : Set := Cons : A -> Stream A -> Stream A.
- This way of defining coinductive types is not well-behaved.
- Use the following codata type instead:

```
codata Stream(a) { hd : a, tl : Stream(a) }
```

Sketch 2

Codata for Functions



A class can be seen as a bundle of functions, which have state. Therefore, the type of functions is nothing but a class with only one method, which we call ap. Applying the function means to execute the method ap. Therefore, if A and B are Java types, we define the type of functions from A to B, $A \to B$, as the following interface (we use the valid Java identifier A_B instead of $A \to B$):

```
interface A_B{B ap(A x);};
```

If f is of type $A \to B$, and a is of type A, then f.ap(a) is the result of applying f to a, for which one might introduce the abbreviation f(a).

Java as a Functional Programming Language (A. Setzer)

Codata for Functions

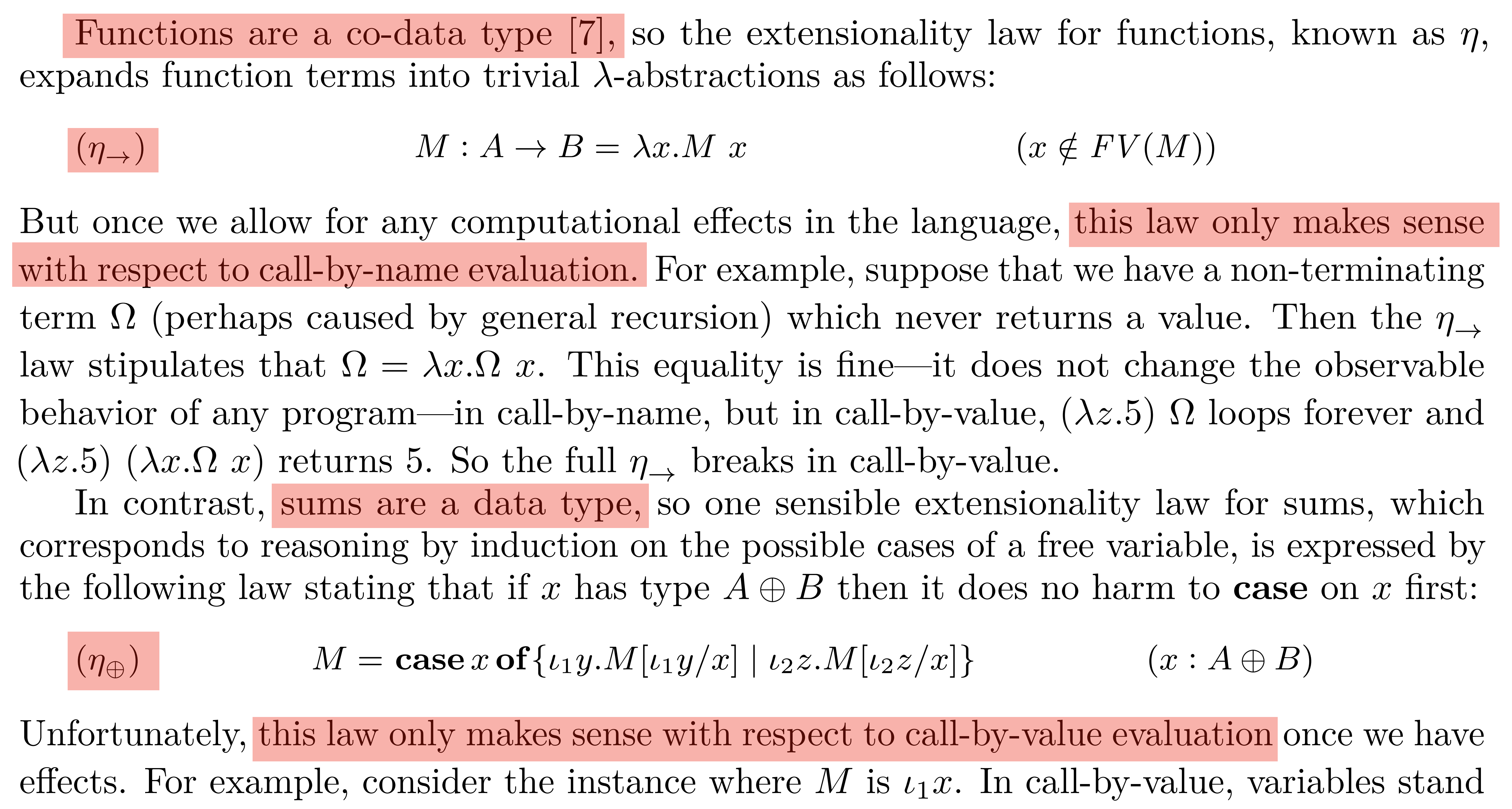
The codata type Fun

- Functional languages usually don't allow to define the function type.
- Codata types are more general than function types.
- Use the following codata type instead:

```
codata Fun(a,b) { ap(x : a) : b }
```

Sketch 3

Codata for Non-Strict Types



Functions are a co-data type [7], so the extensionality law for functions, known as η, expands function terms into trivial λ-abstractions as follows:

$$(\eta_{\to}) \qquad M : A \to B = \lambda x.M\ x \qquad (x \notin FV(M))$$

But once we allow for any computational effects in the language, this law only makes sense with respect to call-by-name evaluation. For example, suppose that we have a non-terminating term Ω (perhaps caused by general recursion) which never returns a value. Then the $\eta_{\to}$ law stipulates that $\Omega = \lambda x.\Omega\ x$. This equality is fine—it does not change the observable behavior of any program—in call-by-name, but in call-by-value, $(\lambda z.5)\ \Omega$ loops forever and $(\lambda z.5)\ (\lambda x.\Omega\ x)$ returns 5. So the full $\eta_{\to}$ breaks in call-by-value.

In contrast, sums are a data type, so one sensible extensionality law for sums, which corresponds to reasoning by induction on the possible cases of a free variable, is expressed by the following law stating that if x has type $A \oplus B$ then it does no harm to **case** on x first:

$$(\eta_{\oplus}) \qquad M = \mathbf{case}\, x\, \mathbf{of}\, \{\iota_1 y.M[\iota_1 y/x] \mid \iota_2 z.M[\iota_2 z/x]\} \qquad (x : A \oplus B)$$

Unfortunately, this law only makes sense with respect to call-by-value evaluation once we have effects. For example, consider the instance where M is $\iota_1 x$. In call-by-value, variables stand

Beyond Polarity: Towards a Multi-Discipline Intermediate Language With Sharing (Downen, Ariola)

Codata for Non-Strict Types

The codata type LPair

- η-Laws for codata types are only valid under call-by-name
- η-Laws for data types are only valid under call-by-value
- Haskell's encoding of lazy pairs is therefore wrong; use the following codata type instead:

```
codata LPair(a,b) { fst: a, snd: b }
```

Sketch 4

Codata for Modalities

Menu (price 17 Euros)

Quiche or Salad
Chicken or Fish
Banana or "Surprise du Chef"*

(*) either "*Profiteroles*" or "*Tarte Tatin*"

$$17E \vdash \left\{ \begin{array}{c} (Q \& S) \\ \otimes \\ (C \& F) \\ \otimes \\ (B \& (P \oplus T)) \end{array} \right.$$

Introduction to Linear Logic and Ludics, Part I (Pierre-Louis Curien)

Codata for Modalities

The data type Tensor and the codata type With

- In linear logic there are two types of logical "and": Tensor ⊗ and With &
- In order to use $A \otimes B$ you have to use both A and B exactly once.
- In order to use $A \,\&\, B$ you have to use A once or use B once.

```
data Tensor(a,b) { Tup(a,b) }
codata With(a,b) { fst: a, snd: b }
```

Sketch 5

Codata for Extensibility

Defunctionalization and refunctionalization establish a correspondence between first-class functions and pattern matching, but the correspondence is not symmetric: Not all uses of pattern matching can be automatically refunctionalized to uses of higher-order functions. To remedy this asymmetry, we generalize from first-class functions to arbitrary codata. This leads us to full defunctionalization and refunctionalization between a codata language based on copattern matching and a data language based on pattern matching.

We observe how programs can be written as matrices so that they are modularly extensible in one dimension but not the other. In this representation, defunctionalization and refunctionalization correspond to matrix transposition which effectively changes the dimension of extensibility a program supports. This suggests applications to the expression problem.

Automatic Refunctionalization to a Language with Copattern Matching: With Applications to the Expression Problem (Rendel, Trieflinger, Ostermann)

Codata for Extensibility

The codata type Exp

- Data types are easily extendable by new consumers which pattern match.
- It is hard to extend data types with new producers / constructors.
- Codata types are easily extendable by new producers which copattern match.
- It is hard to extend codata types with new observations / destructors.

```
data Exp { Lit(Int), Add(Exp,Exp) }
codata Exp { print : String, eval : Int }
```

Why I Care About Codata Types

Five sketches of an elephant

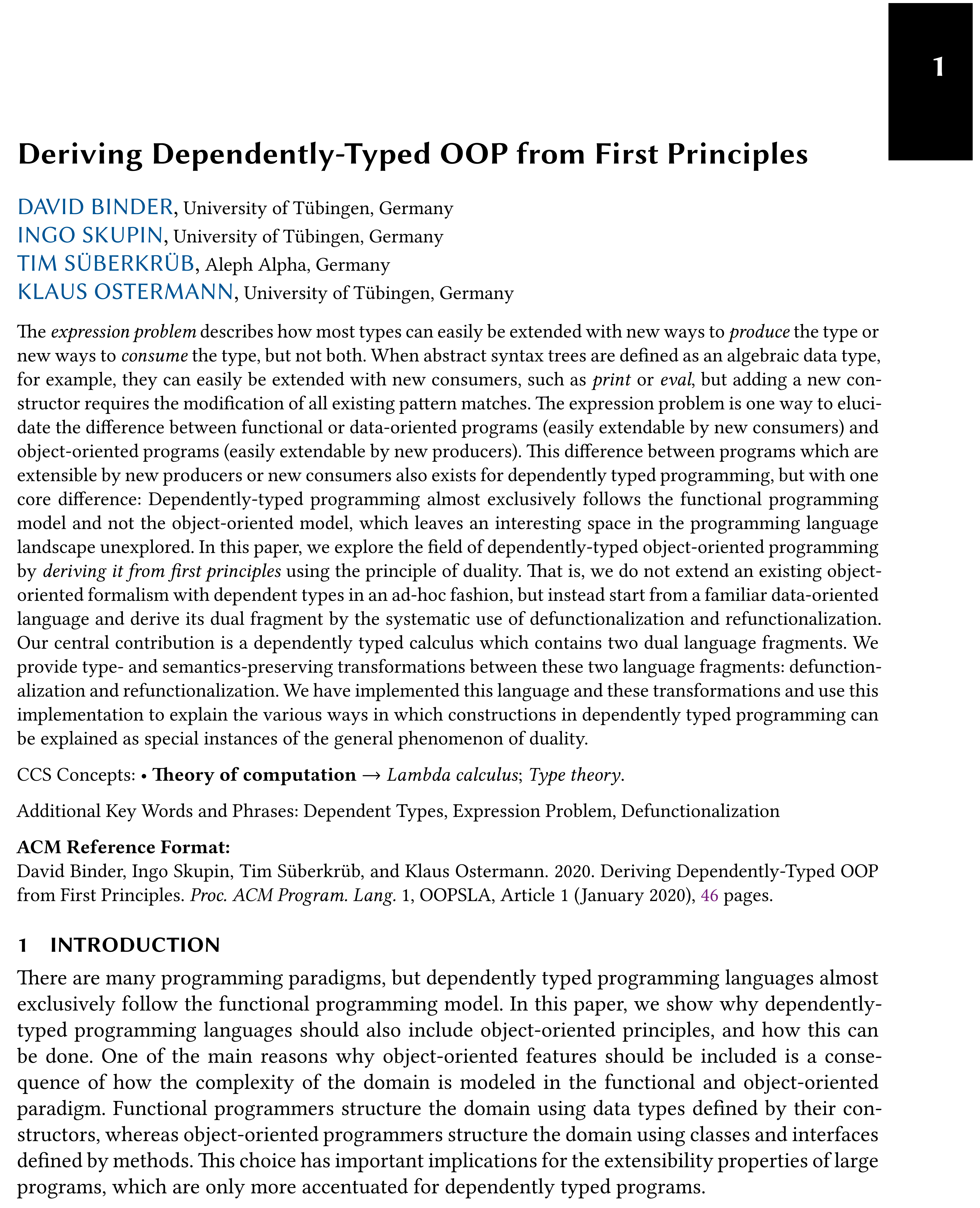
- Sketch *1:* Codata for Infinite Data (**codata Stream**)
- Sketch *2:* Codata for Functions (**codata Fun**) Also dependent functions
- Sketch *3*: Codata for Non-Strict Types (**codata LPair**) Also shift types
- Sketch *4:* Codata for Modalities (**data Tensor vs codata With**) Also $\oplus$, $\otimes$, ! and ?
- Sketch *5:* Codata for Extensibility (**codata Exp**)

Dependent (Co)Data Types

Online demo: polarity-lang.github.io/oopsla24

Deriving Dependently-Typed OOP from First Principles

- Jointly with: Ingo Skupin, Tim Süberkrüb and Klaus Ostermann
- Published at OOPSLA '24
- Preprint: arxiv.org/abs/2403.06707
- Implementation available at: polarity-lang.github.io



1

Deriving Dependently-Typed OOP from First Principles

DAVID BINDER, University of Tübingen, Germany
INGO SKUPIN, University of Tübingen, Germany
TIM SÜBERKRÜB, Aleph Alpha, 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.

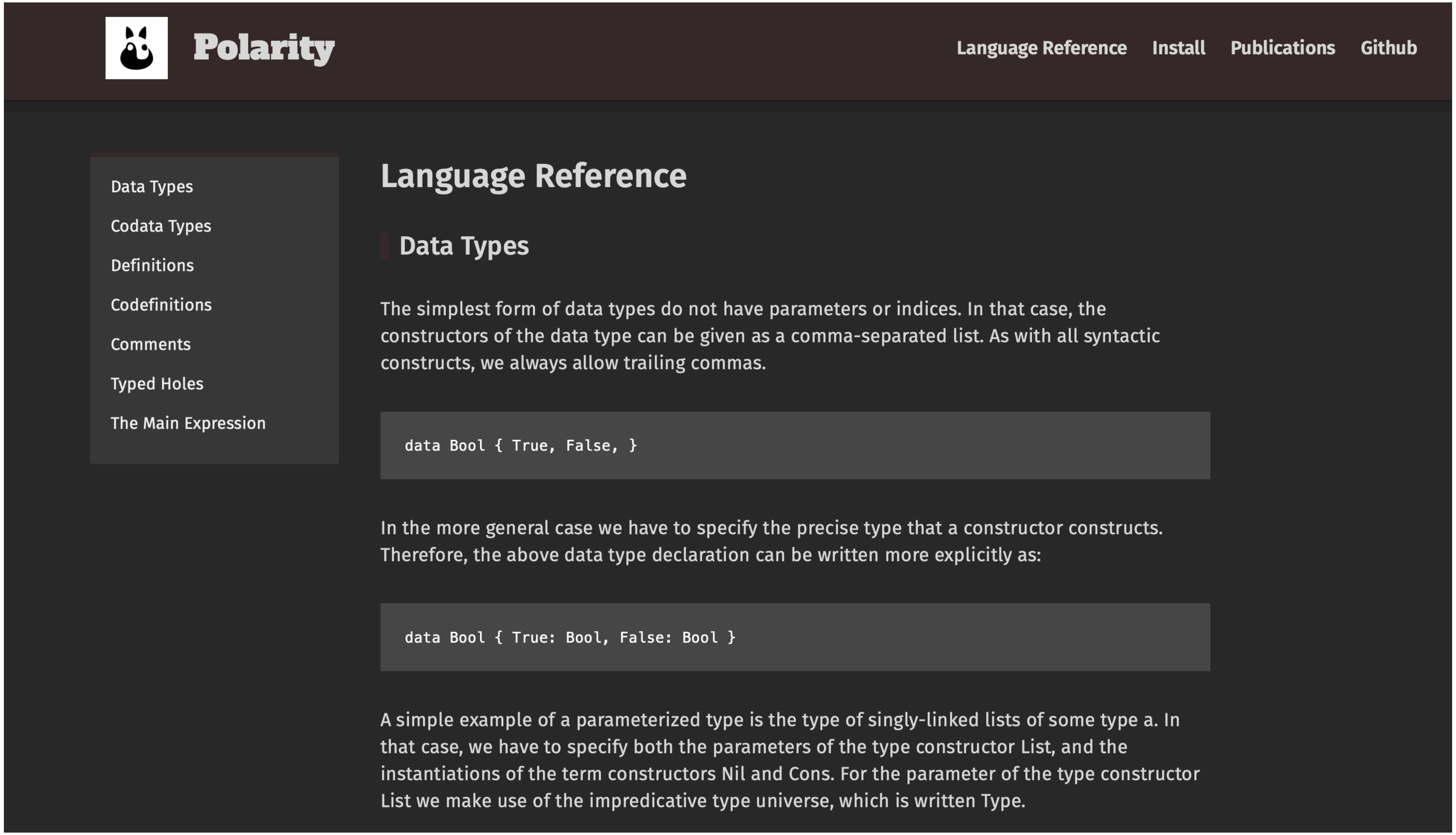
CCS Concepts: • **Theory of computation** → *Lambda calculus*; *Type theory*.

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

ACM Reference Format:
David Binder, Ingo Skupin, Tim Süberkrüb, and Klaus Ostermann. 2020. Deriving Dependently-Typed OOP from First Principles. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 1 (January 2020), 46 pages.

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 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.


Polarity

Language Reference Install Publications Github

Language Reference

Data Types

The simplest form of data types do not have parameters or indices. In that case, the constructors of the data type can be given as a comma-separated list. As with all syntactic constructs, we always allow trailing commas.

```
data Bool { True, False, }
```

In the more general case we have to specify the precise type that a constructor constructs. Therefore, the above data type declaration can be written more explicitly as:

```
data Bool { True: Bool, False: Bool }
```

A simple example of a parameterized type is the type of singly-linked lists of some type a. In that case, we have to specify both the parameters of the type constructor List, and the instantiations of the term constructors Nil and Cons. For the parameter of the type constructor List we make use of the impredicative type universe, which is written Type.

How Does a Dependently-Typed Language with only User-Defined Data and Codata Look Like?

Type Theory based on Dependent Inductive and Coinductive Types

Henning Basold
Radboud University
CWI, Amsterdam
h.basold@cs.ru.nl

Herman Geuvers
Radboud University
Technical University Eindhoven
herman@cs.ru.nl

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 }
```

Copattern Matching!

Booleans

De/Refunctionalization is Matrix transposition!

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

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

Bool	*True*	*False*
neg	False	True

Booleans with Proofs

```
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) }
```

```
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) }
```

Self Parameters!

Dependent Functions

```
-- | 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) }
```

Basold & Geuvers show how to do it without non-dependent functions!

Dependent Pairs

```
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 }
```

Weak and strong Sigma-Types!

Natural Numbers

```
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)) }


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 }
```

Obtain functional encodings (Church, Scott, Parigot) via refunctionalization!

Natural Numbers (continued)

Fu & Stump: Self Types for Dependently Typed Lambda Encodings

```
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 }
```

```
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)) }
```

We had to make some compromises

Design Constraints

The Cost of Total Defunctionalization and Refunctionalization

- No eta-laws for types which will be de/refunctionalized
- Alpha-equivalence cannot be used; comatches are generative $\lambda x . x \neq \lambda y . y$
- All arguments of a type constructor are indices; no parameters
- We use the "Type : Type" axiom
- Positivity is not preserved under de/refunctionalization, so we don't check it

Future Work

Future Work

- Make the language usable: Modules, implicit arguments, code generation,...
- Investigate the combination of dependent types with modalities (à la QTT, Granule)
- Find of consistency checks that are stable under defunctionalization and refunctionalization.

Original Title: "The Proof Expression Problem"

Reviewers didn't like it (for good reason!); still expresses our ambition.

Time for Questions!