From FP to OOP and Back, Consistently

UNSOUND '24

David Binder

University of Tübingen

This was not supposed to be a remote talk :(

This is Work in Progress

- Joint work with Ingo Skupin, Tim Süberkrüb and Klaus Ostermann
- Check out my talk on Wednesday!
- Let's start with a look at the work we want to build upon

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

Part I: A fresh look on FP vs OOP

My Working (Type-Theoretic) Definition of OOP

- Codata Types (Interfaces)
- Subtyping
- Open Recursion

← I will concentrate on this aspect
Cp. William Cook on Data Abstraction

Booleans: The FP Version (I)

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

Bool defined as a data type

Observations defined by pattern matching

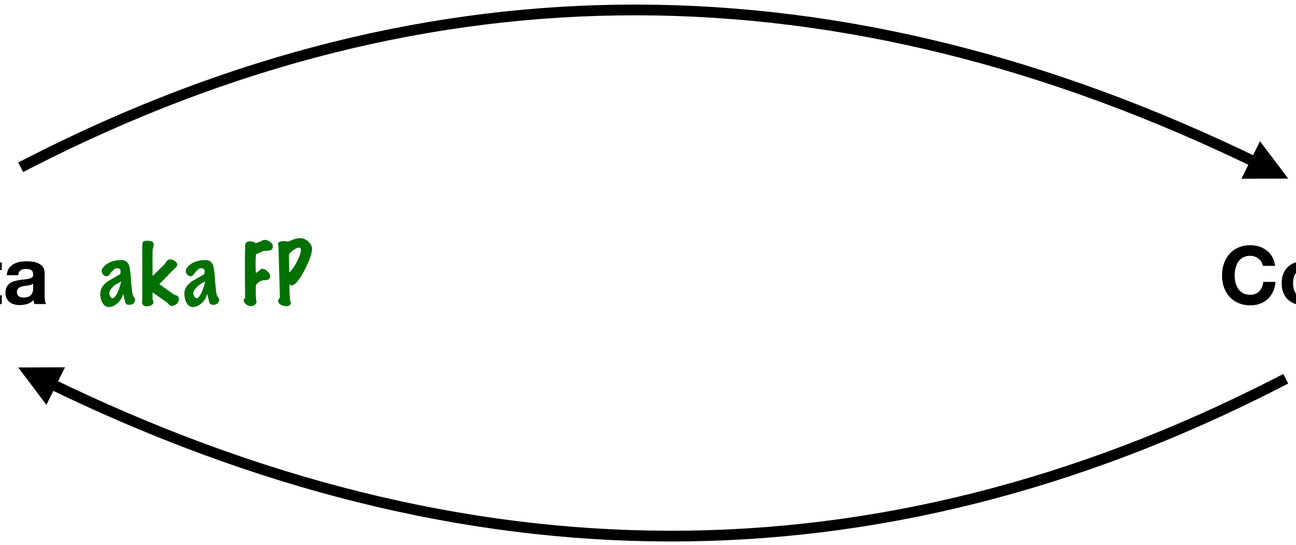
Booleans: The OOP Version (II)

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

Bool defined as a codata type

Inhabitants defined by copattern matching
(implementing an interface)

Refunctionalization



Data aka FP

Codata aka OOP

Defunctionalization

Functions are just a specific instance of a codata type

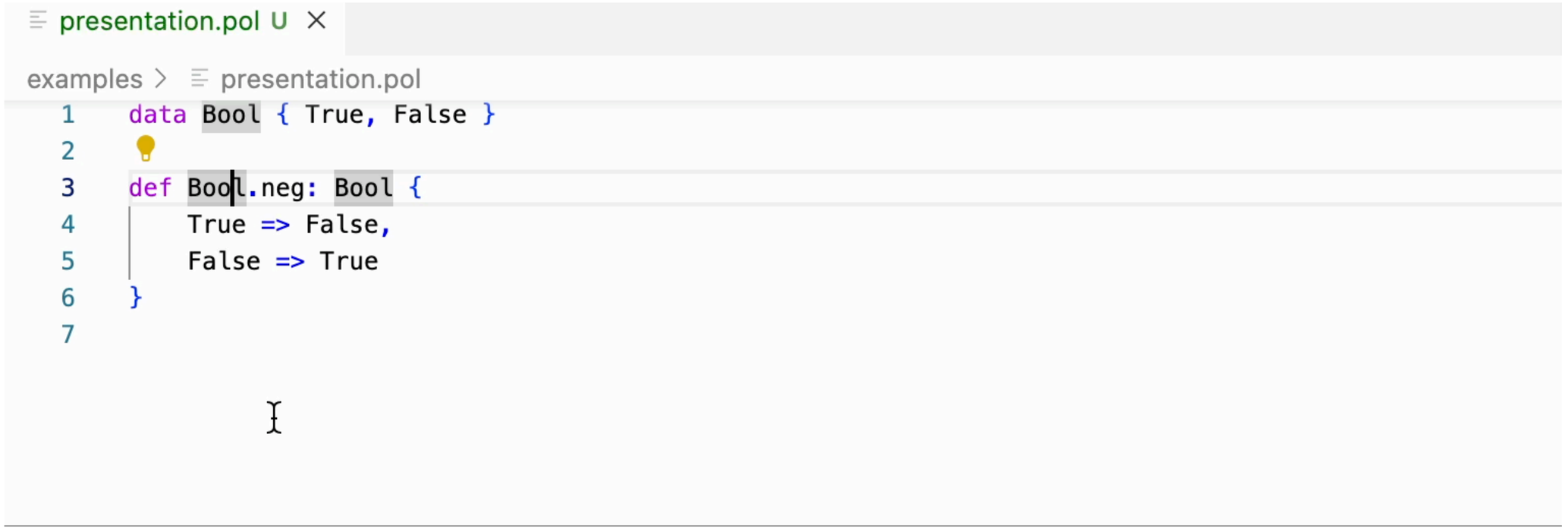
```
data Bool { True, False }
def Bool.neg: Bool {
    True => False,
    False => True }

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

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

Bool	*True*	*False*
neg	False	True

Programs as matrices

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


presentation.pol U
examples
presentation.pol
1 data Bool { True, False }
2
3 def Bool.neg: Bool {
4 True => False,
5 False => True
6 }
7

Now with Dependent Types

Booleans: The FP Version (II)

```
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) }
```

Martin-Löf Equality

Proof that negation is involutive

Booleans: The OOP Version (II)

```
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) }
```

Observations with self parameters

Objects come with correctness proofs

example.pol

```
1   data Eq(a: Type, x y: a) {
2       Refl(a: Type, x: a) : Eq(a, x, x)
3   }
4   data Bool { True, False }
5
6   def Bool.neg: Bool {
7       True => False,
8       False => True
9   }
10
11  def (self: Bool).neg_eq: Eq(Bool, self, self.neg.neg) {
12      True => Refl(Bool, True),
13      False => Refl(Bool, False)
14  }
15
```

What we have achieved in the paper

- No builtin types: Non-dependent and dependent function types are user-defined codata types
- Proof of type soundness
- De-/Refunctionalization is total and type-preserving
- We can de-/refunctionalize types that occur in indizes of type constructors and which are normalized and compared during type checking
- Restrictions on the beta and eta-equality that we can use during type checking.

Part II: Troubles with Consistency

We started with a different title

"The Proof Expression Problem"

Reviewers rightfully rejected that title

```
-- | Expressions of the object language
data Exp {
    -- | Variables using a deBruijn representation
    Var(x: Nat),
    -- | Lambda abstractions
    Lam(body: Exp),
    -- | Function applications
    App(lhs: Exp, rhs: Exp)
}
```

```
-- | Expressions of the object language
codata Exp {
  (e: Exp).weaken_cons(ctx: Ctx, t1 t2: Typ)
      : HasType(ctx, e, t2) -> HasType(ctx.append(Cons(t1, Nil)), e, t2),
  (e: Exp).progress(t: Typ): HasType(Nil, e, t) -> Progress(e),
  (e1: Exp).preservation(e2: Exp, t: Typ)
      : HasType(Nil, e1, t) -> Eval(e1, e2) -> HasType(Nil, e2, t),
  -- | Substituting an expression for a variable in an expression.
  .subst(v: Nat, by: Exp): Exp,
  (e: Exp).subst_lemma(ctx1 ctx2: Ctx, t1 t2: Typ, by_e: Exp)
      : HasType(ctx1.append(Cons(t1, ctx2)), e, t2) -> HasType(Nil,
                                                               by_e,
                                                               t1) -> HasType(ctx1.append(ctx2),
                                                                              e.subst(ctx1.len,
                                                                                      by_e),
                                                                              t2)
}
```

examples > stlc.pol

```
1   ------------------
2   -- Specification --
3   ------------------
4
5   -- | Expressions of the object language
6   data Exp {
7       -- | Variables using a deBruijn representation
8       Var(x: Nat),
9       -- | Lambda abstractions
10      Lam(body: Exp),
11      -- | Function applications
12      App(lhs: Exp, rhs: Exp)
13  }
14
15  -- | Types of the object language
16  data Typ {
17      -- | Function type
18      FunT(t1 t2: Typ),
19      VarT(x: Nat),
20  }
21
22  -- | Typing contexts.
23  -- | Because we use de Bruijn indices the typing context does not contain variable names.
24  data Ctx {
25      -- | The empty context
26      Nil,
27      -- | Adding a typed binding to the context
28      Cons(t: Typ, ts: Ctx),
29  }
30
31  -- | Appending two contexts
32  def Ctx.append(other: Ctx): Ctx {
33      Nil => other,
34      Cons(t, ts) => Cons(t, ts.append(other))
35  }
```

We think we are very close!

Extremely powerful refactorings for proof engineers!

Alas, the system is inconsistent :(

Inconsistent: Every type is inhabitated.

Three Sources of Inconsistency

- We use the Type-in-Type Axiom
- No checks for positivity of data and codata type declarations
- No termination and productivity checks

We could just implement what other proof assistents implement.

But de-/refunctionalized programs would no longer check :(

Problem 1: Universe Hierarchies

Universe Hierarchies

- We currently use the Type-in-Type Axiom, which is known to be inconsistent thanks to Girard.
- We therefore need to introduce some hierarchy of type universes.
- There are unsolved problems with annotating correct universe levels after defunctionalization.
- Cp. Huang & Yallop 2023: Defunctionalization with Dependent Types

Problem 2: Positivity of Type Declarations

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

Negative Occurrence

But this type can be the result of refunctionalizing an ordinary Agda or Coq program using only strictly positive data types!

The binary methods problem from the OOP literature

Problem 3: Termination and Productivity

```
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)) }
```

← 1. Check the type declaration.

← 2. Check termination of plus.

← 3. Check termination of mul.
Assume totality of plus!

There is a clear order in which we check for totality of function definitions.

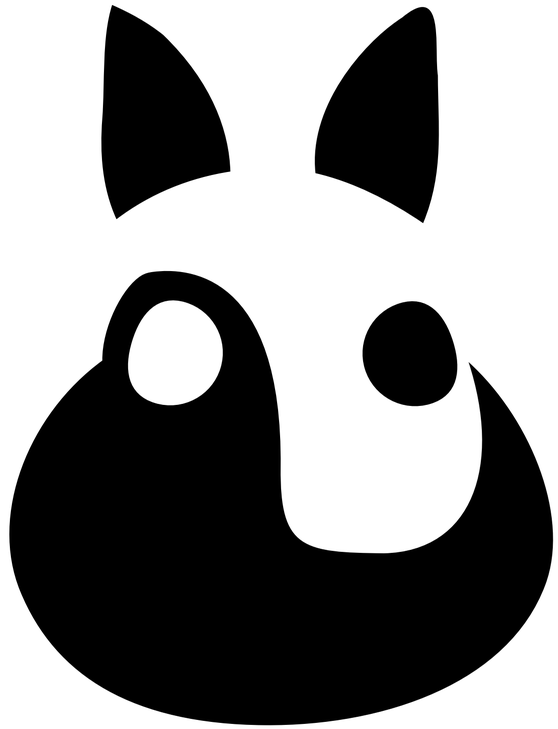
```
codata Nat { plus(n: Nat): Nat, mul(n: Nat): Nat }
codef S(x: Nat): Nat {
    plus(n) => S(x.plus(n)),
    mul(n) => n.plus(x.mul(n)) }
codef Z: Nat {
    plus(n) => n,
    mul(n) => Z }
```

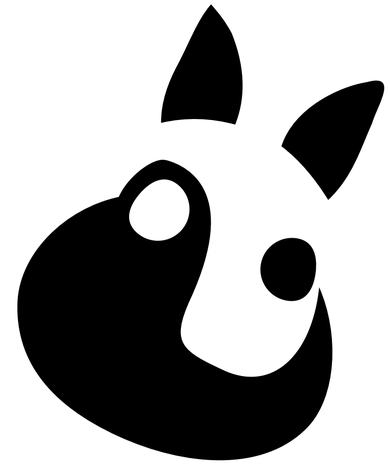
How do we declare the dependency of mul on plus if S and Z are mutually recursive?

We must somehow first construct "partial" objects S and Z for which only plus is defined.

Follow the Development

- polarity-lang.github.io
- Webdemo!
- MIT/Apache Licensed
- Distinguished Artifact @ OOPSLA





Thanks for your attention!