#### **Deriving Dependently-Typed OOP from First Principles OOPSLA '24, Pasadena**

**David Binder, Ingo Skupin, Tim Süberkrüb, Klaus Ostermann** 

University of Tübingen



#### My Working (Type-Theoretic) Definition of OOP

- Codata Types (Interfaces)
- Subtyping
- Open Recursion / Late Binding

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



#### **Booleans: The FP Version (I)**

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

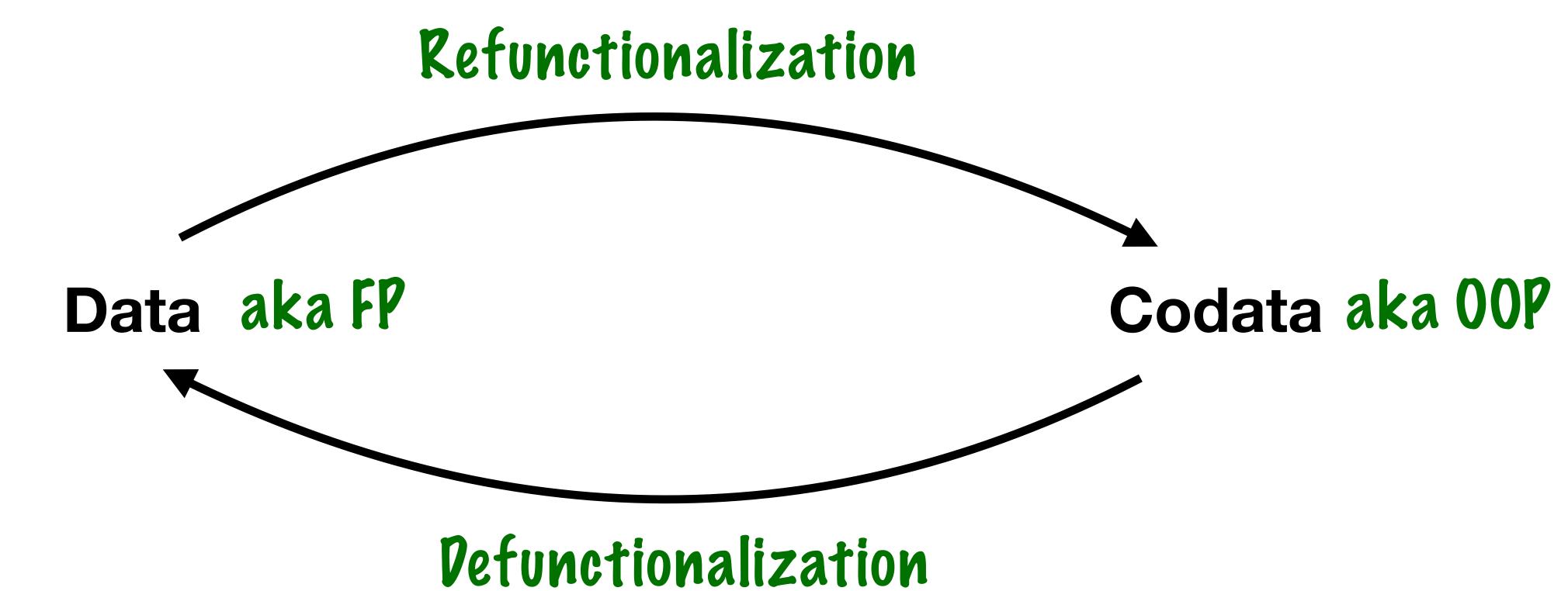
#### data Bool { True, False } - Bool defined as a data type Observations defined by pattern matching



#### **Booleans: The OOP Version (II)**

- codata Bool { neg: Bool } Bool defined as a codata type codef True: Bool { neg => False } codef False: Bool { neg => True }
  - Inhabitants defined by copattern matching (implementing an interface)





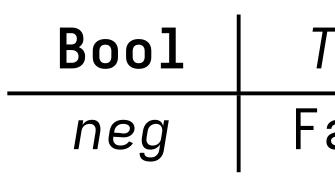
#### De-/Refunctionalization as a principled mechanism to derive symmetric language fragments.

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 }



codata Bool { neg: Bool }

True	False	Programs as matrices
alse	True	

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



```
First presentation.pol U X
examples > First 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)** Martin-Löf Equality 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) }

#### 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) }
```

### Methods with self parameters

#### Objects come with correctness proofs

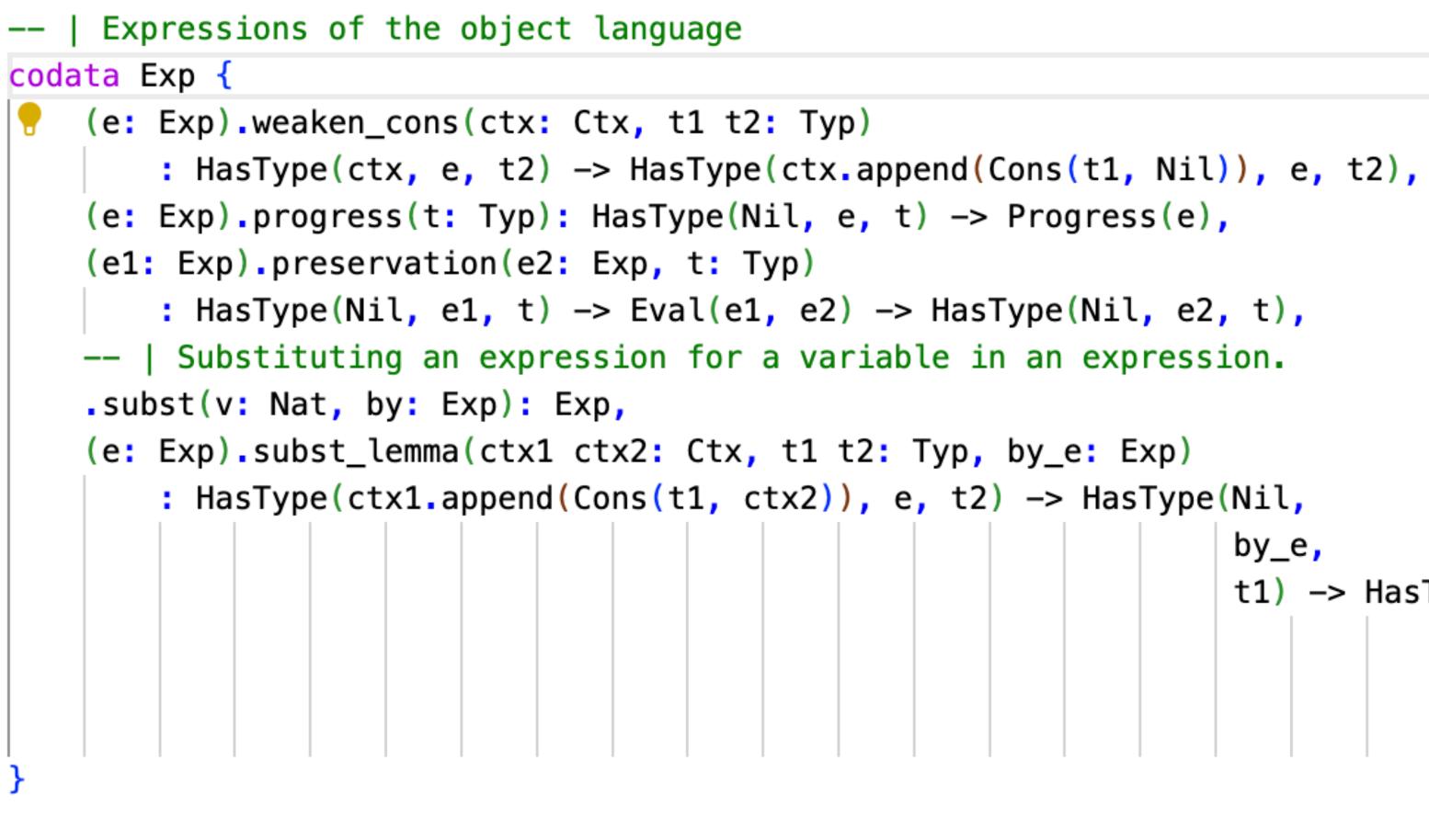


```
= example.pol
     data Eq(a: Type, x y: a) {
 1
         Refl(a: Type, x: a) : Eq(a, x, x)
 2
 3
     } 🥊
     data Bool { True, False }
 4
 5
     def Bool.neg: Bool {
 6
         True => False,
 7
         False => True
 8
 9
10
11
     def (self: Bool).neg_eq: Eq(Bool, self, self.neg.neg) {
12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

## With Applications to the Expression Problem

```
-- | 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)
```

#### Proof of type soundness proceeds by induction on Exp Difficult to extend with new expression nodes



#### Expressions as interface for all theorems that must hold

```
by_e,
t1) -> HasType(ctx1.append(ctx2),
               e.subst(ctx1.len,
                       by_e),
               t2)
```

```
examples \geq \equiv stlc.pol
        -- Specification --
   3
        -- | Expressions of the object language
   5
        data Exp {
   6
            -- | Variables using a deBruijn representation
   7
            Var(x: Nat),
   8
            -- | Lambda abstractions
   9
           Lam(body: Exp),
  10
            -- | Function applications
  11
            App(lhs: Exp, rhs: Exp)
  12
  13
  14
        -- | Types of the object language
  15
       data Typ {
  16
            -- | Function type
  17
           FunT(t1 t2: Typ),
  18
  19
           VarT(x: Nat),
  20
        }
  21
        -- | Typing contexts.
  22
  23
       data Ctx {
  24
            -- | The empty context
  25
  26
           Nil,
  27
            -- | Adding a typed binding to the context
            Cons(t: Typ, ts: Ctx),
  28
  29
  30
        -- | Appending two contexts
  31
        def Ctx.append(other: Ctx): Ctx {
  32
           Nil => other,
  33
  34
            Cons(t, ts) => Cons(t, ts.append(other))
35
```

-- | Because we use de Bruijn indices the typing context does not contain variable names.



More Examples!

#### **Functions are User-Defined**

-- Non-dependent Functions "a -> b" is syntactic sugar for "Fun(a,b)" codata Fun(a b: Type) { Fun(a, b).ap(a b: Type, x: a): b } -- Dependent Functions codata  $\Pi(a: Type, p: a -> Type)$  {  $\Pi(a, p).dap(a: Type, p: a -> Type, x: a): p.ap(a, Type, x) \}$ 

#### **Defined by function application "dap"**

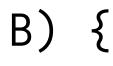
#### Cp. Setzer 2003: Java as a Functional Programming Language

#### **Positive and Negative Pairs**

# data ×<sub>+</sub>(A B: Type) { Pair(A B: Type, x: A, y: B): ×<sub>+</sub>(A, B) } def ×<sub>+</sub>(A, B).π<sub>1</sub>(A B: Type): A { Pair(\_, \_, x, y) => x } def ×<sub>+</sub>(A, B).π<sub>2</sub>(A B: Type): B { Pair(\_, \_, x, y) => y }

### Defined by pairing constructor Corresponds to $\otimes$ in Linear Logic

#### Defined by projections Corresponds to & in Linear Logic



#### Weak and Strong Sigma Types

```
data \Sigma_+(A: Type, T: A \rightarrow Type) \{

Pair(A: Type,

T: A \rightarrow Type,

x: A,

w: T.ap(A, Type, x) )

: \Sigma_+(A, T) \}

def \Sigma_+(A, T).\pi_1(A: Type, T: A \rightarrow Type): A \{

Pair(A, T, x, w) => x }

def (self: \Sigma_+(A, T)).\pi_2(A: Type, T: A \rightarrow Type)

: T.ap(A, Type, self.\pi_1(A, T)) {

Pair(A, T, x, w) => w }
```

#### Linked by De-/Refunctionalization

```
codata \Sigma_{-}(A: Type, T: A \rightarrow Type) \{

\Sigma_{-}(A, T).\pi_{1}(A: Type, T: A \rightarrow Type): A,

(self: \Sigma_{-}(A, T)).\pi_{2}(A: Type, T: A \rightarrow Type)

: T.ap(A, Type, self.\pi_{1}(A, T)) \}

codef Pair(A: Type,

T: A -> Type,

x: A,

w: T.ap(A, Type, x) )

: \Sigma_{-}(A, T) \{

\pi_{1}(A, T) => x,

\pi_{2}(A, T) => w \}
```

#### What we have achieved in the paper

- Dependent type theory with no builtin types: Non-dependent and dependent function types are user-defined codata types
- Proof of type soundness (Extended version @ ArXiV)
- 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
- Various examples: Strong vs. weak Σ-types, codata encodings of natural numbers, dependently-typed programming examples

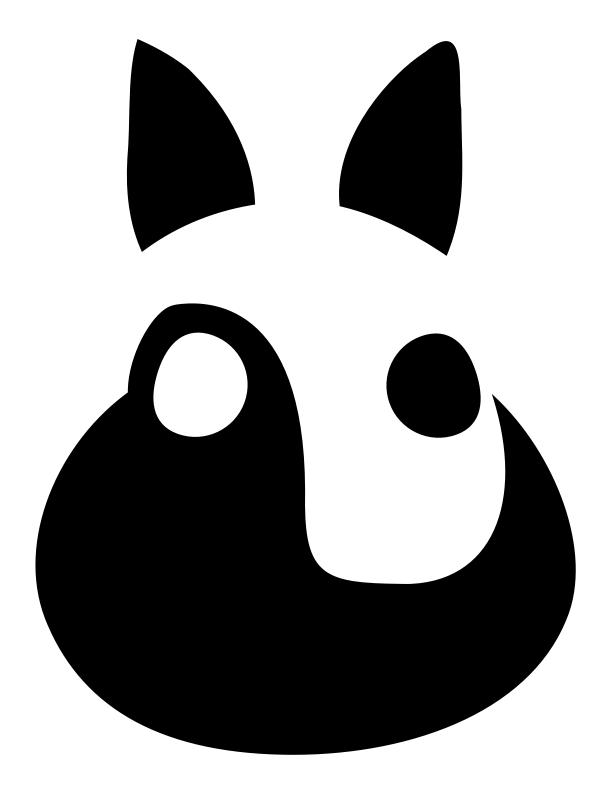
#### What we have \*not\* achieved in the paper

- System is sound, but not consistent
- We use the Type : Type axiom
- We don't check for (strict) positivity of recursive (co-)data declarations
- We don't check for termination or productivity
- Eta-equalities not valid for typechecking
- Some restrictions on judgemental equality

**Difficult to preserve these properties under de-/refunctionalization** 

#### Implementation

- polarity-lang.github.io/oopsla24/
- Implemented in Rust
- LSP Server and VSCode Extension
- All examples run in the browser!
- Actively hacked on :)



## Please like & subscribe: polarity-lang.github.io

