David Binder, Ingo Skupin, Tim Süberkrüb, Klaus Ostermann University of Tübingen

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

1

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) the program results in the property in Figure 1 and 20 minutes in this representation, negation, negation is t
In this representation, negation is the single ob-

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

data Bool $\{$ True, False $\}$ \leftarrow Bool defined as a data type Observations defined by pattern
matching

Booleans: The OOP Version (II) this program results in the probability in the program in Figure 1. In this representation, negative $\sum_{i=1}^n \frac{1}{i!}$ servation of a codata type, and False are definition that \mathbf{r}

- codata Bool $\{$ neg: Bool $\}$ \leftarrow Bool defined as a codata type codef True: Bool { neg => False } codef False: Bool { neg => True }
- itants defined by copattern match Fig. 1. Two representatives of the same program. 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 } data Bool { True, False } servation of a codata type, and the codata type, and the codata type are defined as objects interface. The code
True \approx False.

-
-

(a) Functional programming style. codata Bool { neg: Bool } codef False: Bool { neg => True } codef True: Bool { neg => False }

data Bool { True, False } $True \Rightarrow False,$ type as a matrix. Te two programs of Figure 1, for example, can be represented by the following def Bool.neg: Bool { True => False, False => True } $True > False,$


```
\c{oodata} Bool \{neg\} Rool \}codef False: Bool { neg => True }
            codata Bool { neg: Bool }
            codef True: Bool { neg => False }
```

```
\equiv presentation.pol U \timesexamples \geq \equiv presentation.pol
         data Bool \{ True, False \}1\overline{2}T
         def Bool.neg: Bool {
   \overline{3}True \Rightarrow False,
   4
               False \implies True5
   6
   \overline{7}I
```
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, $False \Rightarrow True \}$ 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) lated more directly as an elimination on a named boolean self which yields a proof that self which yields a proof **educans: 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

```
\equiv example.pol
       data Eq(a: Type, x y: a) {
 \mathbf{1}Refl(a: Type, x: a) : Eq(a, x, x)
 \overline{2}\overline{3}\}data Bool { True, False }
 4
 5
       def Bool.neg: Bool {
 6
            True \Rightarrow False,
  \overline{7}False \Rightarrow True8
 9
10
11
       def (self: Bool).neg_eq: Eq(Bool, self, self.neg.neg) {
12True \Rightarrow Refl(Bool, True),
13
            False \Rightarrow Refl(Bool, False)
14
       \mathcal{F}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
   \overline{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
  17
            -- | Function type
  18
            FunT(t1 t2: Typ),19
            VarT(x: Nat),
  20
        \mathcal{F}21
        -- | Typing contexts.
  2223
        data Ctx {
  24
  25
            -- | The empty context
  26
            Nil,
  27-- | Adding a typed binding to the context
            Cons(t: Typ, ts: Ctx),
  28
  29
        \mathcal{F}30
        -- | Appending two contexts
  31
        def Ctx.append(other: Ctx): Ctx {
  32
            Nil \Rightarrow other,33
            Cons(t, ts) => Cons(t, ts.append(other))
  3435
```
-- | Because we use de Bruijn indices the typing context does not contain variable names.

More Examples!

s_n as also useful. Apart from radical, and we think it is also useful. Apart from reducing the reducing to n cp. Setzer zuus. Java as a functional frogramming fanguage Cp. Setzer 2003: Java as a Functional Programming Language

Defined by function application "dap" Defined by function application "dap"

Functions are User-Defined in a library over a library. Theory

-- | Non-dependent Functions codata Fun(a b: Type) { Fun(a, b).ap(a b: Type, $x: a)$: b } -- | Dependent Functions codata $\Pi(a: Type, p: a \rightarrow Type)$ { $\Pi(a, p)$.dap(a: Type, p: a -> Type, x: a): p.ap(a, Type, x) } $\mathbb T$ both dependent and non-dependent functions should be user-definitions should be user-definitions should be user-"a -> b" is syntactic sugar for "Fun(a,b)"

Tese two representations can be obtained from each other by defunctionalization and refunctionone constructor Pair which takes the type family , and a witness and a witness and a witness and a witness Defined hy nairing constructor **can be obtained from the properties** Corresponds to ⊗ in Linear Logic $\qquad \qquad$ Corresponds to & in Linear Logic $\qquad \qquad$

Positive and Negative Pairs the fraction in its return in its return type; weak 2-types, by contrast, and the fraction in the following contrast, and the following contrast, and the following contrast, and the contrast, and the contrast, and the con one constructor which pairs the frame second element. The frame second element is more obvious in which we have the friends of the fraction in its return types, and the fraction of the following projection in the following $\mathbf{D}_{\mathbf{A}}$ **one construction we construct and second element. The framework and second element. The framework of we construct and second element. The framework is more obvious in the framework in the framework is more obvious in the**

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 } 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 }

\n
$$
\text{codata } x_-(A \ B: \text{Type}) \ \{ \ x_-(A, B) \cdot \pi_1(A \ B: \text{Type}): A, \ x_-(A, B) \cdot \pi_2(A \ B: \text{Type}): B \ \}
$$
\n

\n\n $\text{codef } \text{Pair}(A \ B: \text{Type}, \ x: A, y: B): x_-(A, B) \ \{ \pi_1(_, _, _) => x, \ \pi_2(_, _, _) => y \ \}$ \n

belined by pairing constructor we we we generally projections Corresponds to \otimes in Linear Logic alization. The case when we get we generally we generally project to the Sense of the S-type. Similar to the Z-Defined by pairing constructor Defined by projections

Weak and Strong Sigma Types Weak and Strong Sigma Types the two takes the two family \sim guardians. As a constitution of non-dependent in the case of non-dependent in the case of non-dependent in the case of non-Wheels and Strong Sigme Types the two states the two states the type \sim guments. As a codata type, we still have two projections ¹ and ² as in the case of non-dependent

```
data \Sigma_+(A: Type, T: A \rightarrow Type) \Pair(A: Type,
Pair(A: Type,
           T: A -> Type,
T: A -> Type,
           x: A,
x: A,
           w: T.ap(A, Type, x) )
w: T.ap(A, Type, x) )
        : \Sigma_+(A, T) }
def Σ+(A, T).π<sub>1</sub>(A: Type, T: A -> Type): A {
    Pair(A, T, x, w) => xdef (self: Σ₊(A, T)).π₂(A: Type, T: A -> Type)
def (self: Σ₊(A, T)).π₂(A: Type, T: A -> Type)
    : T.ap(A, Type, self.π<sub>1</sub>(A, T)) {
    Pair(A, T, x, w) => w
```
was not originally designed with codata types in mind, and its codata types in mind, and its codata types are implemented on \mathcal{C} top of dependent records, which conducts when the conduction of conductions are possible. For example, the conductions are possible to conduct the conductions are possible. For example, the conductions are possible. For ex \mathbf{R} is designed with codata types in mind, and its codata types in \mathbf{R} types are implemented on \mathbf{R} top of depending the dependent records, which is a conditioner than \mathbf{r}_i Linked by De-/Refunctionalization

```
codata \Sigma_{-}(A: Type, T: A \rightarrow Type) {
     \Sigma_{-}(A, T).\pi_1(A: Type, T: A -> Type): A,
     (self: \Sigma_{-}(A, T)) \cdot \pi_2(A: Type, T: A \rightarrow Type): T.ap(A, Type, self.π₁(A, T)) }
: T.ap(A, Type, self.π₁(A, T)) }
codef Pair(A: Type,
codef Pair(A: Type,
                  T: A -> Type,
T: A -> Type,
                  x: A,
x: A,
                  w: T.ap(A, Type, x) )
w: T.ap(A, Type, x) )
     :\Sigma_{-}(A, T) {
     \pi_1(A, T) \implies x,\pi_2(A, T) \implies 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

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

Implementation

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

