De-/Refunctionalization with Dependent Data and CodataTypes

UMass Lowell, 2024

David Binder, Ingo Skupin, Tim Süberkrüb, Klaus Ostermann

University of Tübingen

The Core Idea

Booleans: The FP Version (I)

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

← Bool defined as a data type

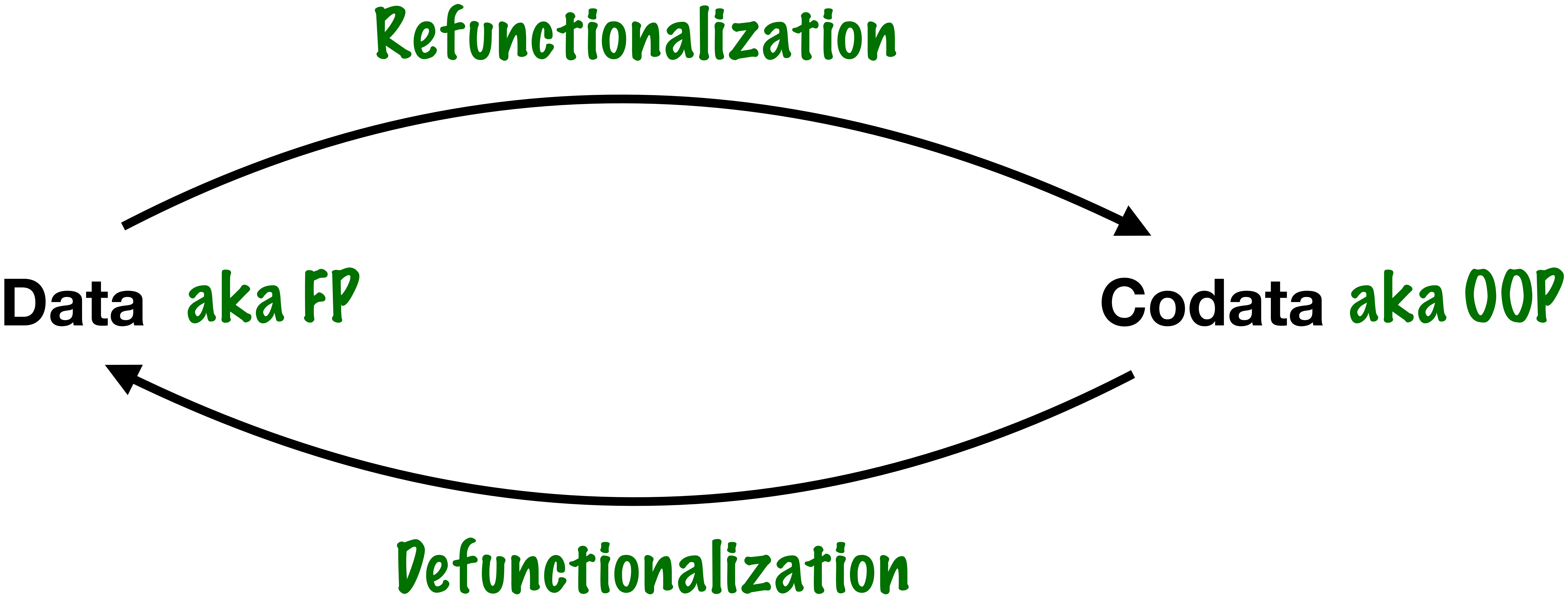
← Observations defined by pattern matching

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)



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 }
```

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

```
data Bool { True, False }

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

```

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) }
```

Methods with self parameters

Objects come with correctness proofs

example.pol

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

```

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 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)
}
```

Expressions as interface for all theorems that must hold

examples > stlc.pol

```
-------------------
-- Specification --
-------------------

-- | 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)
}

-- | Types of the object language
data Typ {
    -- | Function type
    FunT(t1 t2: Typ),
    VarT(x: Nat),
}

-- | Typing contexts.
-- | Because we use de Bruijn indices the typing context does not contain variable names.
data Ctx {
    -- | The empty context
    Nil,
    -- | Adding a typed binding to the context
    Cons(t: Typ, ts: Ctx),
}

-- | Appending two contexts
def Ctx.append(other: Ctx): Ctx {
    Nil => other,
    Cons(t, ts) => Cons(t, ts.append(other))
}
```

More Examples!

Functions are User Defined

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).

A. Setzer, 2003: Java as a Functional Programming Language

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

Functions are User-Defined

```
-- | 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) }
```

"a -> b" is syntactic sugar for "Fun(a,b)"

↑

Defined by dependent function application "dap"

Positive and Negative Pairs

```
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 }
```

Defined by pairing constructor

Corresponds to ⊗ in Linear Logic

```
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 }
```

Defined by projections

Corresponds to & in Linear Logic

Weak and Strong Sigma Types

```
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 }
```

Linked by De-/Refunctionalization

Q: Are there also two ways to define functions?

A: Yes! R. Garner: On the strength of dependent products in the type theory of Martin-Löf.

A more traditional OOP example

```
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
```

```
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. Jacobs, 1995: Objects and classes, coalgebraically

Codata Encodings of Natural Numbers

The Church Encoding

```
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 }
```

The Fu-Stump Encoding

```
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 }
```

Abbreviation for Induction Step

```
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)) }
```

Fu & Stump, 2014: Self Types for Dependently-Typed Lambda Encodings

Problems

Judgemental Equality and Consistency are Difficult

Judgemental Equality must be Preserved (I)

```
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)
```

✓

```
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)
```

✗

Our solution: First program doesn't typecheck.

Judgemental Equality must be Preserved (II)

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

✓

```
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);
```

✗

Our solution: Eta is not valid during equality checking.

Three reasons why our system is inconsistent

- We use the Type : Type axiom instead of a universe hierarchy
- We do not check that recursive occurrences in data and codata declarations are strictly positive
- We do not check for termination or productivity

Difficult to preserve these properties under de-/refunctionalization

Problems appear already in the simply-typed version

Sometimes it is good to be negative!

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

Original motivation for Nakano's guarded logic / guarded type theory.

Well-known problem in theoretical OOP literature.

Conclusion

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

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