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) ← I will concentrate on this aspect
- Subtyping
- Open Recursion / Late Binding

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

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

```
-- | 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 function application "dap"

Cp. Setzer 2003: Java as a Functional Programming Language

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

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