# Codata In Action

Paul Downen     Zachary Sullivan     Zena M. Ariola     Simon Peyton Jones

April 8, 2019

# What is Codata?

$$\text{codata} = \text{infinite objects} \qquad ?$$

codata $\neq$ infinite objects ✗

codata $\supset$ infinite objects ✓

# Data versus Codata

Definition by constructions

```
data Sum a b where
  Left  : a → Sum a b
  Right : b → Sum a b
```

Definition by observations

```
codata Prod a b where
  First  : Prod a b → a
  Second : Prod a b → b
```

# Where does Codata Come From?

- In theory
  - Logic: computational interpretation of sequent calculus, linear logic, polarization, session types, …
  - Algebra: final coalgebras (dual to initial algebras)
- In practice
  - Object-oriented programming (objects are codata!)
  - Functional programming (first-class functions are codata!)

# What is Codata Good For?

- Key Idea: Programming by Observation
- Many applications of codata
  - Infinite objects and coinduction
  - Decomposing Church encodings
  - Decomposing complex problems with demand-driven programming
  - Abstracting over protocol interfaces and their invariants

# Object-Oriented Church Encodings

# Encoding Booleans by Cases

In codata

```
codata Bool where
  If : Bool → (∀a.a → a → a)

 true.If x y = x
false.If x y = y
```

# Encoding Booleans by Cases

In codata

```
codata Bool where
   If : Bool → (∀a.a → a → a)

 true.If x y = x
false.If x y = y
```

In $\lambda$-calculus

$$Bool = \forall a.a \to a \to a$$

$$true = \lambda x.\lambda y.x$$

$$false = \lambda x.\lambda y.y$$

# Walking Down a Tree

```
data Tree where
  Leaf   : Int → Tree
  Branch : Tree → Tree → Tree

walk : (Int → a) → (a → a → a) → Tree → a
walk b f (Leaf x)     = b x
walk b f (Branch l r) = f (walk b f l)
                          (walk b f r)
```

# Walking Down a Tree with the Visitor Pattern

```
codata TreeVisitor a where
  VisitLeaf   : TreeVisitor a → (Int → a)
  VisitBranch : TreeVisitor a → (a → a → a)

codata Tree where
  Walk : Tree → (∀a. TreeVisitor a → a)

leaf : Int → Tree
(leaf x).Walk v = v.VisitLeaf x

branch : Tree → Tree → Tree
(branch l r).Walk v = v.VisitBranch (l.Walk v)
                                    (r.Walk v)
```

# The Visitor Pattern in $\lambda$-calculus

$$TreeVisitor\ a = (Int \to a) \times (a \to a \to a)$$

$$Tree = \forall a. TreeVisitor\ a \to a$$

$$visitLeaf : TreeVisitor\ a \to Int \to a = fst$$

$$visitBranch : TreeVisitor\ a \to a \to a \to a = snd$$

$$leaf : Int \to Tree$$

$$leaf\ x = \lambda v.\ (visitLeaf\ v)\ x$$

$$branch : Tree \to Tree \to Tree$$

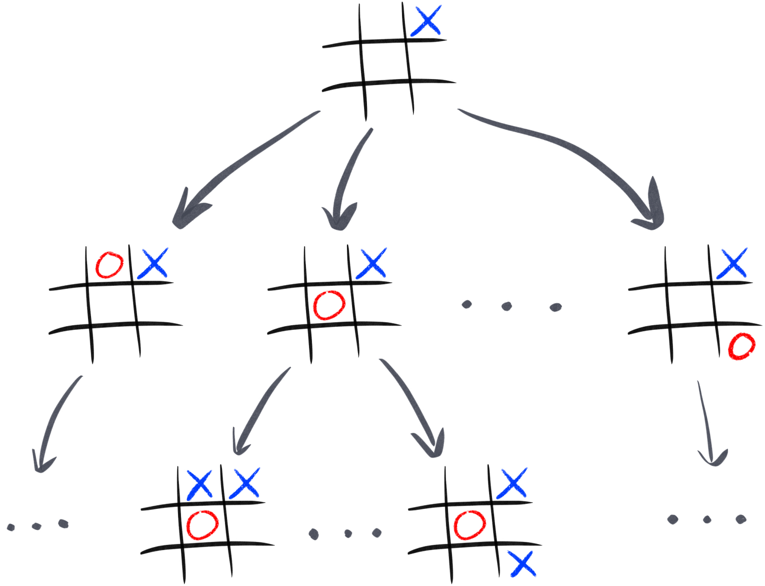
$$branch\ l\ r = \lambda v.\ (visitBranch\ v)\ (l\ a\ v)\ (r\ a\ v)$$

# Demand-Driven Programming

# Why ~~functional~~ Demand-Driven Programming Matters

- Problems should be decomposed into smaller sub-problems
- But sometimes traditional imperative programming prevents decomposition with "one big, messy loop"
- "Why Functional Programming Matters" (Hughes '89) showed how functional programming can help recover decomposition
- Key Idea: Demand-driven programming
- Lazy functional programming is one way to be demand-driven
- Codata is another way, which applies to many more languages

# Let's Play a Game



# The Dream of Decomposition

```
eval : Board → Int
eval = maximize ∘ mapT score ∘ prune 5 ∘ gameTree

gameTree : Board → Tree Board
prune    : Int → Tree a → Tree a
mapT     : (a → b) → Tree a → Tree b
score    : Board → Int
maximize : Tree Int → Int
```

# Decomposition with Codata

```
codata Tree a where
  Node     : Tree a → a
  Children : Tree a → List (Tree a)

gameTree : Board → Tree Board
(gameTree b).Node     = b
(gameTree b).Children = map gameTree (moves b)

prune : Int → Tree a → Tree a
(prune x t).Node     = t.Node
(prune 0 t).Children = []
(prune x t).Children = map (prune(x-1)) t.Children
```

# Interfaces, Abstractions, and Invariants

# Protocol Interface as a Codata Type

```
codata Database a where
  Select : Database a → (a → Bool) → List a
  Delete : Database a → (a → Bool) → Database a
  Insert : Database a → a → Database a
```

# Abstracting Over an Interface

```
copy : Database a → Database a → Database a
copy from to =
  let rows = from.Select(λ_ → True)
  in  foldr (λrow db → db.Insert row) to rows
```

The same client code does many things depending on `Database a` objects

Might copy between different systems (like MySQL, Oracle, etc.)

Might also be a virtual simulations in short-term memory, useful for testing client code as-is

# Protocol Invariants as an Indexed Codata Type

```
index Raw, Bound, Live

codata Socket i where
  Bind    : Socket Raw   → String → Socket Bound
  Connect : Socket Bound → Socket Live
  Send    : Socket Live  → String → ()
  Receive : Socket Live  → String
  Close   : Socket Live  → ()
```

`newSocket().Bind(addr).Send("Hello")` is ill-typed!

Linear types can go further: ensure all sockets are closed once

# Intercompiling Codata and Data

# Visitor Pattern: Data → Codata

Turn this

```
data Foo where
  One   : A → Foo
  Two   : B → Foo
  Three : C → Foo
```

Into that

```
codata FooVisitor r where
  VisitOne   : FooVisitor r → A → r
  VisitTwo   : FooVisitor r → B → r
  VisitThree : FooVisitor r → C → r

codata Foo’ where
  FooCase : ∀ r. FooVisitor r → r
```

# Tabulation: Codata → Data

Turn this

```
codata Foo where
  One   : Foo → A
  Two   : Foo → B
  Three : Foo → C
x : Foo
```

Into that

```
data Foo’ where
  FooTable : A → B → C → Foo’

x’ : Foo’
x’ = FooTable (x.One) (x.Two) (x.Three)
```

# Dependent Products: Codata → Data + $\prod$

Turn this

```
codata Foo where
  One : Foo → A
  Two : Foo → B
x : Foo
```

Into that

```
data FooMessage r where
  One' : FooMessage A
  Two' : FooMessage B
type Foo' = ∀r. FooMessage r → r

x' : Foo'
x' m = case m of One' → x.One
                 Two' → x.Two
```

# A Note on Evaluation Order

- Each compilation is correct for call-by-name and call-by-need
- Call-by-need sharing makes tabulation efficient for free
- Dependent products require explicit sharing (on pain of algorithmic slowdown)
- Call-by-value is also correct with manual intervention
  - Visitor pattern requires A-normalizing constructor arguments
  - Tabulation requires explicit delay/force
  - Dependent products are correct as-is

# A Note on Types

- Compilation applies to untyped terms, but preserves typing
- Different typing complexity for codata $\rightarrow$ data compilations
  - Dependent products requires GADTs
  - Tabulation only requires simple types (but extends to more complex type systems)
- Indexed data and codata types can be compiled by simplifying indexes to type equalities
- Some care is needed to preserve typing of empty objects

# Wrapping it Up

# Lessons Learned

- Codata appears all over the place
- Codata has many practical and theoretical applications
  - But take care: solution $\neq$ problem
  - Codata $\neq$ infinite objects
  - Laziness $\neq$ demand-driven programming
- Codata $\leftrightarrow$ data compilation is straightforward in stock implementations
- Codata is common ground between object-oriented and functional idioms
- Codata is language agnostic (different paradigms, different evaluation orders) and brings techniques to a larger audience

# Status of data and codata today

- Object-oriented languages: an abundance of codata, a scarcity of data
  - Define any codata type you want as an object
  - Only a few built-in primitive data types (integers, booleans, etc.)
- Functional languages: an abundance of data, a scarcity of codata
  - Define any data type you want as a (G)ADT
  - Only one built-in primitive codata type (functions)

Your language should be rich in data and codata, now!