Copatterns
Programming Infinite Objects by Observations

A. Abel[1] B. Pientka[2] D. Thibodeau[2] A. Setzer[3]

[1]Department of Computer Science
Ludwig-Maximilians-University Munich, Germany

[2]School of Computer Science
McGill University, Montreal, Canada

[3]Computer Science
Swansea University, Wales, UK

Principles of Programming Languages
Roma, Italy
23 January 2013

Crash course "Programming in the Infinite"
Final Exam

Crash course "Programming in the Infinite"
Final Exam

Problem 1 (Duality): Complete this table!

finite	infinite
algebra	coalgebra
inductive	coinductive
constructors	destructors
pattern matching	

Approaches to Infinite Structures

1. Just functions. (Scheme, ML)
   - Delay implemented as dummy abstraction, force as dummy application.
   - Memoization needs imperative references.
2. Terminal coalgebras.
   - SymML [Hagino, 1987].
   - Charity [Cockett, 1990s]: Programming with morphism (pointfree).
   - Object-oriented programming: Objects react to messages.
3. Lists/trees of infinite depth.
   - Convenient: program just with pattern matching.
   - Haskell: everything lazy. Finite = infinite.
   - Coq: inductive/coinductive types both via constructors.

Which is best for dependent types?

What's wrong with Coq's CoInductive?

- Coq's coinductive types are non-wellfounded data types.

```
CoInductive Stream : Type :=
| cons (head : nat) (tail : Stream).

CoFixpoint zeros : Stream := cons 0 zeros.
```

- Reduction of cofixpoints only under match.
  Necessary for strong normalization.

$$\begin{aligned}
\textbf{case}\ \text{cons}\ a\ s\ \textbf{of}\ \text{cons}\ x\ y \Rightarrow t &= t[a/x][s/y] \\
\textbf{case}\ \text{cofix}\ f\ \ \textbf{of}\ \ \mathit{branches} &= \textbf{case}\ f\ (\text{cofix}\ f)\ \textbf{of}\ \ \mathit{branches}
\end{aligned}$$

- Leads to loss of subject reduction. [Gimenez, 1996; Oury, 2008]

Issue 1: Loss of Subject Reduction

$\mathsf{Stream} \; : \; \mathsf{Type}$	a codata type
$\mathsf{cons} \; : \; \mathbb{N} \to \mathsf{Stream} \to \mathsf{Stream}$	its (co)constructor
$\mathsf{zeros} \; : \; \mathsf{Stream}$	inhabitant of U
$\mathsf{zeros} \; = \; \mathsf{cofix}\ (\mathsf{cons}\ 0)$	$\mathsf{u} = \mathsf{cons}\ 0\ (\mathsf{cons}\ 0\ (\ldots$
$\mathsf{force} \; : \; \mathsf{Stream} \to \mathsf{Stream}$	an identity
$\mathsf{force}\ s \; = \; \mathbf{case}\ s\ \mathbf{of}\ \mathsf{cons}\ x\ y \Rightarrow \mathsf{cons}\ x\ y$	
$\mathsf{eq} \; : \; (s : \mathsf{Stream}) \to s \equiv \mathsf{force}\ s$	equality type
$\mathsf{eq}\ s \; = \; \mathbf{case}\ s\ \mathbf{of}\ \mathsf{cons}\ x\ y \Rightarrow \mathsf{refl}$	dep. elimination
$\mathsf{eq}_{\mathsf{zeros}} \; : \; \mathsf{zeros} \equiv \mathsf{cons}\ 0\ \mathsf{zeros}$	offending term
$\mathsf{eq}_{\mathsf{zeros}} \; = \; \mathsf{eq}\ \mathsf{zeros} \longrightarrow \mathsf{refl}$	$\not\vdash \mathsf{refl} : \mathsf{zeros} \equiv \mathsf{cons}\ 0\ \mathsf{zeros}$

Analysis

- Problematic: dependent matching on coinductive data.

$$\frac{\Gamma \vdash s : \mathsf{Stream} \qquad \Gamma,\ x : \mathbb{N},\ y : \mathsf{Stream} \vdash t : C(\mathsf{cons}\ x\ y)}{\Gamma \vdash \mathbf{case}\, s\, \mathbf{of}\ \mathsf{cons}\ x\ y \Rightarrow t : C(s)}$$

- [McBride, 2009]: *Let's see how things unfold*.

Issue 2: Deep Guardedness Not Supported

- Fibonacci sequence obeys recurrence:

			0	1	1	2	3	5	8	...
		0	1	1	2	3	5	8	13	...
zipWith (_+_)	0	1	1	2	3	5	8	13	21	...

- Direct recursive definition:

```
fib = cons 0 (cons 1 (zipWith _+_ fib (tail fib)))
fib = cons 0 (              F            (tail fib))
```

- Diverges under Coq's reduction strategy:

```
tail fib
  = F (tail fib)
  = F (F (tail fib))
  = ...
```

Solution: Paradigm shift

Understand coinduction not through construction,

but through observations.

Our contribution:

- New definition scheme "by observation" with copatterns.
- Defining equations hold unconditionally.
- Subject reduction.
- Coverage.
- Strong normalization. (In progress.)

Function Definition by Observation

- A function is a **black box**. We can apply it to an argument (experiment), and observe its result (behavior).
- Application is the defining principle of functions [Granström's dissertation 2009].

$$\frac{f : A \to B \qquad a : A}{f\ a : B}$$

- λ-abstraction is derived, secondary to application.
- Typical semantic view of functions.

Infinite Objects Defined by Observation

- A coinductive object is a **black box**.
- There is a finite set of experiments (projections) we can perform.
- The object is determined by the observations we make.
- Generalize (Agda) records to coinductive types.

```
record Stream : Set where
  coinductive
  field
    head : ℕ
    tail : Stream
```

- `head` and `tail` are the experiments we can make on `Stream`.
- Objects of type `Stream` are defined by the results of these experiments.

Infinite Objects Defined by Observation

- New syntax for defining a cofixpoint.

```
zeros : Stream
head zeros = 0
tail zeros = zeros
```

- Defining the "constructor".

```
cons : ℕ → Stream → Stream
head ((cons x) y) = x
tail ((cons x) y) = y
```

- We call `(head _)` and `(tail _)` **projection copatterns**.
- And `(_ x)` and `(_ y)` **application copatterns**.
- A left-hand side `(head ((_ x) y))` is a **composite** copattern.

Patterns and Copatterns

- Patterns

$$
\begin{array}{lcll}
p & ::= & x & \text{Variable pattern} \\
 & | & () & \text{Unit pattern} \\
 & | & (p_1, p_2) & \text{Pair pattern} \\
 & | & c\ p & \text{Constructor pattern}
\end{array}
$$

- Copatterns

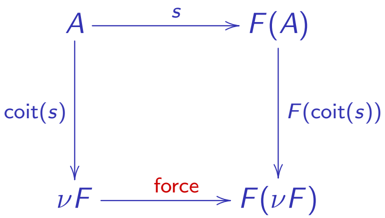
$$
\begin{array}{lcll}
q & ::= & \cdot & \text{Hole} \\
 & | & q\ p & \text{Application copattern} \\
 & | & d\ q & \text{Projection/destructor copattern}
\end{array}
$$

- Definitions

$$
\begin{array}{lcl}
q_1[f/\cdot] & = & t_1 \\
 & \vdots & \\
q_n[f/\cdot] & = & t_n
\end{array}
$$

Category-theoretic Perspective

- Functor F, coalgebra $s : A \to F(A)$.
- Terminal coalgebra $\mathsf{force} : \nu F \to F(\nu F)$ (elimination).
- Coiteration $\mathsf{coit}(s) : A \to \nu F$ constructs infinite objects.

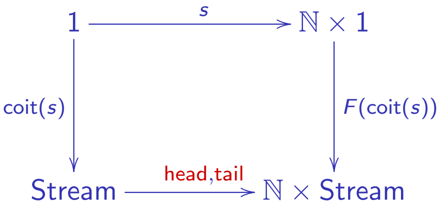


- Computation rule: Only unfold infinite object in elimination context.

$$\mathsf{force}(\mathsf{coit}(s)(a)) = F(\mathsf{coit}(s))(s(a))$$

Instance: Stream

- With $F(X) = \mathbb{N} \times X$ we get the streams $\mathsf{Stream} = \nu F$.
- With $s() = (0, ())$ we get $\mathsf{zeros} = \mathsf{coit}(s)()$.



- Computation: $(\mathsf{head}, \mathsf{tail})(\mathsf{coit}(s)()) = (0, \mathsf{coit}(s)())$.

Deep Copatterns: Fibonacci-Stream

- Fibonacci sequence obeys this recurrence:

$$\begin{array}{r|r} & \begin{array}{cccccccc} 0 & 1 & 1 & 2 & 3 & 5 & 8 & \ldots \\ 1 & 1 & 2 & 3 & 5 & 8 & 13 & \ldots \end{array} & \begin{array}{r} (\mathsf{fib}) \\ (\mathsf{tail}\,\mathsf{fib}) \end{array} \\ \mathsf{zipWith}\ (\_+\_) & \overline{\begin{array}{cccccccc} 1 & 2 & 3 & 5 & 8 & 13 & 21 & \ldots \end{array}} & \mathsf{tail}\,(\mathsf{tail}\,\mathsf{fib}) \end{array}$$

- This directly leads to a definition by copatterns:

```
fib : Stream ℕ
(tail (tail fib)) = zipWith _+_ fib (tail fib)
(head (tail fib)) = 1
(      (head fib)) = 0
```

- Strongly normalizing definition of fib!

Interactive Program Development

- Goal: cyclic stream of numbers.

$$\begin{array}{lcl} \mathsf{cycleNats} & : & \mathbb{N} \to \mathsf{Stream}\ \mathbb{N} \\ \mathsf{cycleNats}\ n & = & n, n-1, \ldots, 1, 0, N, N-1, \ldots, 1, 0, \ldots \end{array}$$

- Fictuous interactive Agda session.

$$\begin{array}{lcl} \mathsf{cycleNats} & : & \mathsf{Nat} \to \mathsf{Stream\ Nat} \\ \mathsf{cycleNats} & = & ? \end{array}$$

- Split result (function).

$$\mathsf{cycleNats}\ x \quad = \quad ?$$

- Split result again (stream).

$$\begin{array}{lcl} \mathsf{head}\ (\mathsf{cycleNats}\ x) & = & ? \\ \mathsf{tail}\ \ \ (\mathsf{cycleNats}\ x) & = & ? \end{array}$$

Interactive Program Development

- Finish first clause:

$$\begin{array}{lll} \text{head}\ (\text{cycleNats}\ x) & = & x \\ \text{tail}\ \ (\text{cycleNats}\ x) & = & ? \end{array}$$

- Split x in second clause.

$$\begin{array}{lll} \text{head}\ (\text{cycleNats}\ x) & = & x \\ \text{tail}\ \ (\text{cycleNats}\ 0) & = & ? \\ \text{tail}\ \ (\text{cycleNats}\ (1 + x')) & = & ? \end{array}$$

- Fill remaining right hand sides.

$$\begin{array}{lll} \text{head}\ (\text{cycleNats}\ x) & = & x \\ \text{tail}\ \ (\text{cycleNats}\ 0) & = & \text{cycleNats}\ N \\ \text{tail}\ \ (\text{cycleNats}\ (1 + x')) & = & \text{cycleNats}\ x' \end{array}$$

Coverage

- Coverage algorithm:
- Start with the trivial covering.
- Repeat
  - split a pattern variable

  until computed covering matches user-given patterns.

Copattern Coverage

- Coverage algorithm:
- Start with the trivial covering. (Copattern · "hole")
- Repeat
  - split result or
  - split a pattern variable

  until computed covering matches user-given patterns.

Deriving Covering Set of Clauses

start		$(\ \vdash \cdot : \mathbb{N} \to \mathsf{Stream})$
split function		$(x{:}\mathbb{N} \vdash \cdot\ x : \mathsf{Stream})$
split stream	$(x{:}\mathbb{N} \vdash \mathsf{head}\ (\cdot\ x) : \mathbb{N})$	$(x{:}\mathbb{N} \vdash \mathsf{tail}\ (\cdot\ x) : \mathsf{Stream})$
split var.	$(x{:}\mathbb{N} \vdash \mathsf{head}\ (\cdot\ x) : \mathbb{N})$	$(\ \vdash \mathsf{tail}\ (\cdot\ 0) : \mathsf{Stream})$ $(x'{:}\mathbb{N} \vdash \mathsf{tail}\ (\cdot\ (1 + x')) : \mathsf{Stream})$

Syntax

finite / positive / type checking			
	type	introduction t	pattern p
tuple	$A_1 \times A_2$	(t_1, t_2)	(p_1, p_2)
data	μ,$+$	$c\ t$	$c\ p$
infinite / negative / type inference			
	type	copattern q	elimination e
function	$A_1 \to A_2$	$q\ p$	$e\ t$
record	ν,&	$d\ q$	$d\ e$

Results

- Subject reduction.
- Non-deterministic coverage algorithm.
- Progress: Any well-typed term that is not a value can be reduced.
- Thus, well-typed programs do not go wrong.
- Prototypic implementations: MiniAgda, Agda.

Suggestion to Haskellers

Use copattern syntax for newtypes!

```
newtype State s a = State { runState :: s -> (a,s) }

instance Monad (State s) where

  runState (return a) s = (a,s)

  runState (m >>= k)  s =
    let (a,s') = runState m
    in  runState (k a) s'
```

Conclusions

- Future work:
  - MiniAgda: A productivity checker with sized types.
  - TODO: Prove strong normalization.
  - TODO: Integrate copatterns into Agda's kernel.
- Related Work:
  - Hagino (1987): Categorical data types.
  - Cockett et al. (1990s): Charity.
  - Zeilberger, Licata, Harper (2008): Focusing sequent calculus.

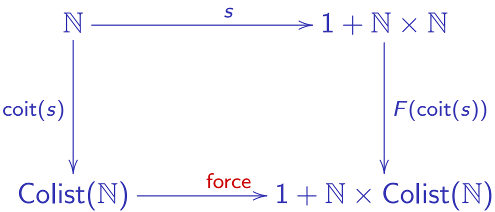
Crash course "Programming in the Infinite"
Model Solution

Problem 1 (Duality): Complete this table!

finite	infinite
algebra	coalgebra
inductive	coinductive
constructors	destructors
pattern matching	**copattern matching**

Instance: Colists of Natural Numbers

- With $F(X) = 1 + \mathbb{N} \times X$ we get $\nu F = \text{Colist}(\mathbb{N})$.
- With $s(n : \mathbb{N}) = \text{inr}(n, n+1)$ we get $\text{coit}(s)(n) = (n, n+1, n+2, \ldots..)$.



Colists in Agda

- Colists as record.

```
data Maybe A : Set where
  nothing :       Maybe A
  just    : A → Maybe A

record Colist A : Set where
  coinductive
  field
    force : Maybe (A × Colist A)
```

- Sequence of natural numbers.

```
nats : ℕ  →  ℕ
force (nats n) = just (n , nats (n + 1))
```

Coverage Rules

$\boxed{A \;\lhd|\; \vec{Q}}$ Typed copatterns $\vec{Q}$ cover elimination of type A.

- Result splitting:

$$\frac{}{A \;\lhd|\; (\vdash \cdot : A)} \qquad \frac{\ldots(\Delta \vdash q : B \to C)\ldots}{\ldots(\Delta, x : B \vdash q\ x : C)\ldots}$$

$$\frac{\ldots(\Delta \vdash q : R)\ldots}{\ldots(\Delta \vdash d\ q : R_d)_{d \in R}\ldots}$$

- Variable splitting:

$$\frac{\ldots(\Delta, x : A_1 \times A_2 \vdash q[x] : C)\ldots}{\ldots(\Delta, x_1{:}A_1, x_2{:}A_2 \vdash q[(x_1, x_2)] : C)\ldots}$$

$$\frac{\ldots(\Delta, x{:}D \vdash q[x] : C)\ldots}{\ldots(\Delta, x'{:}D_c \vdash q[c\ x'] : C)_{c \in D}\ldots}$$

Type-theoretic background

Foundation: coalgebras (category theory) and focusing (polarized logic)

polarity	positive	negative
linear types	$1, \oplus, \otimes, \mu$	$\multimap, \&, \nu$
Agda types	`data`	$\rightarrow$, `record`
extension	finite	infinite
introduction	constructors	definition by copatterns
elimination	pattern matching	message passing
categorical	algebra	coalgebra