Copatterns Programming Infinite Objects by Observations

Andreas Abel

Department of Computer Science Ludwig-Maximilians-University Munich

TCS Oberseminar Ludwig-Maximilians-University Munich 18 January 2013

Copatterns

- Originated from AIM discussions since 2008 on coinduction.
- MiniAgda prototype (March 2011).
- Started on Agda prototype in Nov 2011 (with James Chapman)
- Currently in abstract syntax only
- Goal: integrate into the core (internal syntax)

What's wrong with Coq's Coinductive?

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

```
CoInductive U : Type :=
| inn : U -> U.
CoFixpoint u : U := inn u.
```

 Reduction of cofixpoints is context-sensitive, to maintain strong normalization.

```
case (inn s) of inn y \Rightarrow t = t[s/y]
case (cofix f) of inn y \Rightarrow t = \text{case}(f(\text{cofix } f))) of inn y \Rightarrow t
```

A problem with subject reduction in Coq

 $\begin{array}{cccc} U & : & \mathsf{Type} & & \mathsf{a} \mathsf{\ codata\ type} \\ \mathsf{inn} & : & \mathsf{U} \to \mathsf{U} & & \mathsf{its} \mathsf{\ (co)} \mathsf{constructor} \end{array}$

 $egin{array}{lll} u & : & U & & \mbox{inhabitant of } U \\ u & = & \mbox{cofix inn} & & u = \mbox{inn}(\mbox{inn}(\ldots)) \end{array}$

force : $\mathsf{U} \to \mathsf{U}$ an identity

force = λx . case x of inn $y \Rightarrow \text{inn } y$

eq : $(x : U) \rightarrow x \equiv \text{force } x$ dep. elimination

eq = λx . case x of inn $y \Rightarrow \text{refl}$

 eq_u : $u \equiv inn u$ offending term eq_u = $eq u \longrightarrow refl$ $\not\vdash refl$: $u \equiv inn u$

4 / 32

Analysis

• Problematic: dependent matching on coinductive data.

$$\frac{\Gamma \vdash u : \mathsf{U} \qquad \Gamma, \ y : \mathsf{U} \vdash t : C(\mathsf{inn} \ y)}{\Gamma \vdash \mathsf{case} \ u \ \mathsf{of} \ \mathsf{inn} \ y \Rightarrow t : C(u)}$$

• Solution: Paradigm shift.

Understand coinduction not through construction, but through observations.

 Hinderance: The human mind seems to prefer concrete constructions over abstract black boxes with an ascribed behavior.

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.
- Transfer this to other infinite objects: coinductive things.

Infinite Objects Defined by Observation

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

```
record U : Set where
  coinductive
  field
   out : U
```

- out : U -> U is the experiment we can make on U.
- Objects of type U are defined by the result of this experiment.

Infinite Objects Defined by Observation

Defining a cofixpoint.

```
u : U
out u = u
```

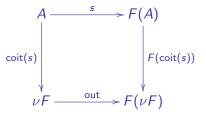
• Defining the "constructor".

```
inn : U -> U
out (inn x) = x
```

- We call (out _) a projection copattern.
- And (_ x) an application copattern.
- The whole thing (out (_ x)) is a composite copattern.

Category-theoretic Perspective

- Functor F, coalgebra $s: A \to F(A)$.
- Terminal coalgebra out : $\nu F \rightarrow F(\nu F)$ (elimination).
- Coiteration $coit(s): A \rightarrow \nu F$ constructs infinite objects.



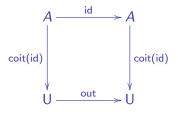
Computation rule: Only unfold infinite object in elimination context.

$$\operatorname{out}(\operatorname{coit}(s)(a)) = F(\operatorname{coit}(s))(s(a))$$

9 / 32

Instance: U

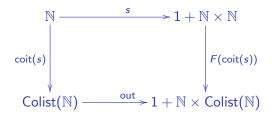
- With F(X) = X we get the coinductive unit type $U = \nu F$.
- With $s = id_A$ we get u = coit(id)(a) for arbitrary a : A.



Computation out(u) = u.

Instance: Colists of Natural Numbers

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



Colists in Agda

Colists as record.

```
data Maybe A : Set where nothing : Maybe A just : A \rightarrow Maybe A
```

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

Sequence of natural numbers.

```
\begin{array}{lll} \text{nats} & : \; \mathbb{N} & \rightarrow \; \; \mathbb{N} \\ \text{out (nats n)} & = \; \text{just (n , nats (n + 1))} \end{array}
```

Streams

Streams have two observations: head and tail.

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

A stream is defined by its head and tail.

```
zipWith : {A B C : Set} -> (A -> B -> C) ->
  Stream A -> Stream B -> Stream C
head (zipWith f as bs) = f (head as) (head bs)
tail (zipWith f as bs) = zipWith f (tail as) (tail bs)
```

Deep Copatterns: Fibonacci-Stream

• Fibonacci sequence obeys this recurrence:

```
0 1 1 2 3 5 8 ... (fib)
1 1 2 3 5 8 13 ... (tail fib)
1 2 3 5 8 13 21 ... tail (tail fib)
```

This directly leads to a definition by copatterns:

```
fib : Stream \mathbb{N} (tail (tail fib)) = zipWith _+_ fib (tail fib) (head (tail fib)) = 1 ( (head fib)) = 0
```

Strongly normalizing definition of fib!

Fibonacci

Definition with cons not strongly normalizing.

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

Diverges under Coq's reduction strategy:

Type-theoretic motivation

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

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

Types

$$A, B, C ::= X$$
 Type variable Positive type Negative type

Type examples

Data types (algebraic types):

```
List A = \mu X \langle \text{nil } 1 \mid \text{cons } (A \times X) \rangle

Nat = \mu X \langle \text{zero } 1 \mid \text{suc } X \rangle

Maybe A = \mu_- \langle \text{nothing } 1 \mid \text{just } A \rangle

0 = \mu_- \langle \rangle (positive empty type)
```

Record types (coalgebraic types):

```
\begin{array}{lcl} \mathsf{Stream} \ A &=& \nu X \{\mathsf{head} : A, \mathsf{tail} : X \} \\ \mathsf{Colist} \ A &=& \nu X \{\mathsf{out} : \mu_{-} \langle \mathsf{nil} \ 1 \mid \mathsf{cons} \ (A \times X) \rangle \} \\ \mathsf{Vector} \ A &=& \nu_{-} \{\mathsf{length} : \mathsf{Nat}, \ \mathsf{elems} : \mathsf{List} \ A \} \\ \top &=& \nu_{-} \{ \} & (\mathsf{negative} \ \mathsf{unit} \ \mathsf{type}) \end{array}
```

Terms

Bidirectional Type Checking

• $\Delta \vdash t \Rightarrow A$ In context Δ , the type of term t is inferred as A.

$$\frac{\Delta(x) = A}{\Delta \vdash f \Rightarrow \Sigma(f)} \qquad \frac{\Delta(x) = A}{\Delta \vdash x \Rightarrow A} \qquad \frac{\Delta \vdash t \Rightarrow \nu XR}{\Delta \vdash t.d \Rightarrow R_d[\nu XR/X]}$$
$$\frac{\Delta \vdash t_1 \Rightarrow A_1 \rightarrow A_2 \qquad \Delta \vdash t_2 \Leftarrow A_1}{\Delta \vdash t_1 \ t_2 \Rightarrow A_2}$$

• $\triangle \vdash t \Leftarrow A$ In context \triangle , term t checks against type A.

$$\frac{\Delta \vdash t \Rightarrow A \qquad A = C}{\Delta \vdash t \Leftarrow C} \qquad \frac{\Delta \vdash t \Leftarrow D_c[\mu XD/X]}{\Delta \vdash c \ t \Leftarrow \mu XD}$$

$$\frac{\Delta \vdash t_1 \Leftarrow A_1 \quad \Delta \vdash t_2 \Leftarrow A_2}{\Delta \vdash (t_1, t_2) \Leftarrow A_1 \times A_2}$$

Patterns and Copatterns

Patterns

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

Copatterns

$$egin{array}{lll} q & ::= & \cdot & & \mbox{Hole} \ & | & q & p & \mbox{Application copattern} \ & | & q.d & \mbox{Destructor copattern} \end{array}$$

Pattern Type Checking

• $\Delta \vdash p \Leftarrow A$ Pattern p checks against type A, yielding Δ .

$$\frac{\Delta \vdash p \Leftarrow D_{c}[\mu XD/X]}{\Delta \vdash c \ p \Leftarrow \mu XD}$$

$$\frac{\Delta \vdash p \Leftarrow D_{c}[\mu XD/X]}{\Delta \vdash c \ p \Leftarrow \mu XD}$$

$$\frac{\Delta_{1} \vdash p_{1} \Leftarrow A_{1} \qquad \Delta_{2} \vdash p_{2} \Leftarrow A_{2}}{\Delta_{1}, \Delta_{2} \vdash (p_{1}, p_{2}) \Leftarrow A_{1} \times A_{2}}$$

• $\Delta \mid A \vdash q \Rightarrow C$ Copattern q eliminates given type A into inferred type C, yielding context Δ .

$$\frac{\Delta \mid A \vdash q \Rightarrow \nu XR}{\Delta \mid A \vdash q.d \Rightarrow R_d[\nu XR/X]}$$

$$\frac{\Delta_1 \mid A \vdash q \Rightarrow B \to C \qquad \Delta_2 \vdash p \Leftarrow B}{\Delta_1, \Delta_2 \mid A \vdash q \mid p \Rightarrow C}$$

22 / 32

Fibonacci Example Program

- Program consists of type signatures Σ and rewrite rules Rules.
- Example entries for fib.

```
\begin{split} & \Sigma(\mathsf{fib}) &= \nu X \{\mathsf{head} : \mu Y \, \langle \mathsf{zero} \ 1 \, | \, \mathsf{suc} \ Y \rangle \,, \ \mathsf{tail} : X \} \\ & \mathsf{Rules}(\mathsf{fib}) = \left\{ \begin{array}{l} \cdot \, .\mathsf{head} & \mapsto \mathsf{zero} \, () \\ \cdot \, .\mathsf{tail} \, .\mathsf{head} & \mapsto \mathsf{suc} \, (\mathsf{zero} \, ()) \\ \cdot \, .\mathsf{tail} \, .\mathsf{tail} & \mapsto \mathsf{zipWith} \,\, \_+\_ \,\, \mathsf{fib} \, \, (\mathsf{fib} \,\, .\mathsf{tail}) \end{array} \right\} \end{split}
```

Evaluation

- Redexes have form E[f].
- Evaluation contexts E.

$$E ::= \cdot \qquad \text{Hole}$$

$$\mid E e \qquad \text{Application}$$

$$\mid E.d \qquad \text{Projection}$$

• To reduce redex, we need to match *E* against copatterns *q*.

(Co)pattern Matching

• $|t| = p \setminus \sigma$ Term t matches with pattern p under substitution σ .

$$\frac{t = ? p \searrow \sigma}{c \ t = ? c \ p \searrow \sigma}$$

$$\frac{t = ? p \searrow \sigma}{c \ t = ? c \ p \searrow \sigma}$$

$$\frac{t_1 = ? p_1 \searrow \sigma_1 \qquad t_2 = ? p_2 \searrow \sigma_2}{(t_1, t_2) = ? (p_1, p_2) \searrow \sigma_1, \sigma_2}$$

• $E = q \setminus \sigma$ Evaluation context E matches copattern q returning substitution σ .

$$\frac{E = {}^{?} q \searrow \sigma}{E.d = {}^{?} q.d \searrow \sigma}$$

$$\frac{E = {}^{?} q \searrow \sigma \qquad t = {}^{?} p \searrow \sigma'}{E t = {}^{?} q p \searrow \sigma, \sigma'}$$

Interactive Program Development

Goal: cyclic stream of numbers.

```
cycleNats : \mathbb{N} \to \mathsf{Stream} \ \mathbb{N} cycleNats n = n, n-1, \ldots, 1, 0, N, N-1, \ldots, 1, 0, \ldots
```

Fictuous interactive Agda session.

```
 \begin{array}{lll} \mathsf{cycleNats} & : & \mathsf{Nat} \to \mathsf{Stream} \ \mathsf{Nat} \\ \mathsf{cycleNats} & = & ? \end{array}
```

Split result (function).

cycleNats
$$x = ?$$

Split result again (stream).

```
head (cycleNats x) = ? tail (cycleNats x) = ?
```

Interactive Program Development

Last state:

```
head (cycleNats x) = ? tail (cycleNats x) = ?
```

• Split x in second clause.

```
head (cycleNats x) = ?
tail (cycleNats 0) = ?
tail (cycleNats (1 + x')) = ?
```

Fill right hand sides.

```
head (cycleNats x) = x
tail (cycleNats 0) = cycleNats N
tail (cycleNats (1 + x')) = cycleNats x'
```

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.

Coverage Rules

 $A \triangleleft |\vec{Q}|$ Typed copatterns \vec{Q} cover elimination of type A.

Result splitting:

$$\frac{A \triangleleft | \overrightarrow{Q} (\Delta \vdash q \Rightarrow B \rightarrow C)}{A \triangleleft | \overrightarrow{Q} (\Delta, x : B \vdash q x \Rightarrow C)}$$

$$\frac{A \triangleleft | \overrightarrow{Q} (\Delta, x : B \vdash q x \Rightarrow C)}{A \triangleleft | \overrightarrow{Q} (\Delta \vdash q \Rightarrow \nu XR)}$$

$$\frac{A \triangleleft | \overrightarrow{Q} (\Delta \vdash q \Rightarrow \rho XR)}{A \triangleleft | \overrightarrow{Q} (\Delta \vdash q . d \Rightarrow R_d[\nu XR/X])_{d \in R}}$$

Variable splitting:

$$\frac{A \triangleleft | \vec{Q} (\Delta, x : A_1 \times A_2 \vdash q \Rightarrow C)}{A \triangleleft | \vec{Q} (\Delta, x_1 : A_1, x_2 : A_2 \vdash q[(x_1, x_2)/x] \Rightarrow C)}$$

$$\frac{A \triangleleft | \vec{Q} (\Delta, x : \mu XD \vdash q \Rightarrow C)}{A \triangleleft | \vec{Q} (\Delta, x' : D_c[\mu XD/X] \vdash q[c \ x'/x] \Rightarrow C)_{c \in D}}$$

Results

- Subject reduction.
- Progress: Any well-typed term that is not a value can be reduced.
- Thus, well-typed programs do not go wrong.

Future Work

- A productivity checker with sized types.
- Proof of strong normalization.

Conclusions

Accepted for presentation at POPL 2013:

Abel, Pientka, Thibodeau, and Setzer Copatterns – Programming Infinite Structures by Observation.

- Related Work:
 - Cockett et al. (1990s): Charity.
 - Zeilberger, Licata, Harper (2008): Focusing sequent calculus.