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.

$$
\begin{array}{lcl}
\mathsf{case}\,(\mathsf{inn}\ s)\quad \mathsf{of}\ \mathsf{inn}\ y \Rightarrow t & = & t[s/y] \\
\mathsf{case}\,(\mathsf{cofix}\ f)\,\mathsf{of}\ \mathsf{inn}\ y \Rightarrow t & = & \mathsf{case}\,(f\ \ (\mathsf{cofix}\ f))\,\mathsf{of}\ \mathsf{inn}\ y \Rightarrow t
\end{array}
$$

A problem with subject reduction in Coq

U	$:$	Type	a codata type
inn	$:$	$\mathsf{U} \to \mathsf{U}$	its (co)constructor
u	$:$	U	inhabitant of U
u	$=$	$\mathsf{cofix\ inn}$	$\mathsf{u} = \mathsf{inn}(\mathsf{inn}(\dots$
force	$:$	$\mathsf{U} \to \mathsf{U}$	an identity
force	$=$	$\lambda x.\, \mathsf{case}\, x\, \mathsf{of\ inn}\ y \Rightarrow \mathsf{inn}\ y$	
eq	$:$	$(x : \mathsf{U}) \to x \equiv \mathsf{force}\ x$	dep. elimination
eq	$=$	$\lambda x.\, \mathsf{case}\, x\, \mathsf{of\ inn}\ y \Rightarrow \mathsf{refl}$	
eq_{u}	$:$	$\mathsf{u} \equiv \mathsf{inn}\ \mathsf{u}$	offending term
eq_{u}	$=$	$\mathsf{eq}\ \mathsf{u} \longrightarrow \mathsf{refl}$	$\not\vdash \mathsf{refl} : \mathsf{u} \equiv \mathsf{inn}\ \mathsf{u}$

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 $\mathsf{out} : \nu F \to F(\nu F)$ (elimination).
- Coiteration $\mathsf{coit}(s) : A \to \nu F$ constructs infinite objects.

$$\begin{tikzcd} A \arrow[r, "s"] \arrow[d, "\mathsf{coit}(s)"'] & F(A) \arrow[d, "F(\mathsf{coit}(s))"] \\ \nu F \arrow[r, "\mathsf{out}"] & F(\nu F) \end{tikzcd}$$

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

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

Instance: U

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

$$\begin{tikzcd} A \arrow[r, "\mathsf{id}"] \arrow[d, "\mathsf{coit}(\mathsf{id})"'] & A \arrow[d, "\mathsf{coit}(\mathsf{id})"] \\ \mathsf{U} \arrow[r, "\mathsf{out}"] & \mathsf{U} \end{tikzcd}$$

- Computation $\mathsf{out}(\mathsf{u}) = \mathsf{u}$.

Instance: Colists of Natural Numbers

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

$$
\begin{tikzcd}
\mathbb{N} \arrow[r, "s"] \arrow[d, "\mathrm{coit}(s)"'] & 1 + \mathbb{N} \times \mathbb{N} \arrow[d, "F(\mathrm{coit}(s))"] \\
\mathrm{Colist}(\mathbb{N}) \arrow[r, "\mathrm{out}"] & 1 + \mathbb{N} \times \mathrm{Colist}(\mathbb{N})
\end{tikzcd}
$$

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
    out : Maybe (A × Colist A)
```

- Sequence of natural numbers.

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

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 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:

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

Type-theoretic motivation

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

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

Types

$$
\begin{array}{lll}
A, B, C & ::= X & \text{Type variable} \\
& \mid P & \text{Positive type} \\
& \mid N & \text{Negative type} \\
\\
P & ::= 1 & \text{Unit type} \\
& \mid A \times B & \text{Cartesian product} \\
& \mid \mu X D & \text{Data type} \\
\\
N & ::= A \to B & \text{Function type} \\
& \mid \nu X R & \text{Record type} \\
\\
D & ::= \langle c_1\ A_1 \mid \cdots \mid c_n\ A_n \rangle & \text{Variant (labeled sum)} \\
R & ::= \{d_1 : A_1, \ldots, d_n : A_n\} & \text{Record (labeled product)}
\end{array}
$$

Type examples

- Data types (algebraic types):

$$
\begin{array}{lcl}
\mathsf{List}\ A & = & \mu X\,\langle \mathsf{nil}\ 1 \mid \mathsf{cons}\ (A \times X)\rangle \\
\mathsf{Nat} & = & \mu X\,\langle \mathsf{zero}\ 1 \mid \mathsf{suc}\ X\rangle \\
\mathsf{Maybe}\ A & = & \mu_{\_}\langle \mathsf{nothing}\ 1 \mid \mathsf{just}\ A\rangle \\
0 & = & \mu_{\_}\langle\rangle \qquad\qquad\qquad \text{(positive empty type)}
\end{array}
$$

- Record types (coalgebraic types):

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

Terms

e, t, u	$::=$	f	Defined symbol (e.g. function)
	$\mid$	x	Variable
	$\mid$	$()$	Unit (empty tuple)
	$\mid$	(t_1, t_2)	Pair
	$\mid$	$c\ \ t$	Constructor application
	$\mid$	$t_1\ \ t_2$	Application
	$\mid$	$t.d$	Destructor application

Bidirectional Type Checking

- $\boxed{\Delta \vdash t \Rightarrow A}$ In context Δ, the type of term t is inferred as A.

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

$$\frac{\Delta \vdash t_1 \Rightarrow A_1 \to A_2 \qquad \Delta \vdash t_2 \Leftarrow A_1}{\Delta \vdash t_1\ t_2 \Rightarrow A_2}$$

- $\boxed{\Delta \vdash t \Leftarrow A}$ In context Δ, term t checks against type A.

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

$$\frac{}{\Delta \vdash () \Leftarrow 1} \qquad \frac{\Delta \vdash t_1 \Leftarrow A_1 \qquad \Delta \vdash t_2 \Leftarrow A_2}{\Delta \vdash (t_1, t_2) \Leftarrow A_1 \times A_2}$$

Patterns and Copatterns

- Patterns

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

- Copatterns

$$
\begin{array}{rcll}
q & ::= & \cdot & \text{Hole} \\
  & \mid & q\ p & \text{Application copattern} \\
  & \mid & q.d & \text{Destructor copattern}
\end{array}
$$

Pattern Type Checking

- $\boxed{\Delta \vdash p \Leftarrow A}$ Pattern p checks against type A, yielding Δ.

$$\frac{}{x : A \vdash x \Leftarrow A} \qquad \frac{\Delta \vdash p \Leftarrow D_c[\mu X D/X]}{\Delta \vdash c\ p \Leftarrow \mu X D}$$

$$\frac{}{\vdash () \Leftarrow 1} \qquad \frac{\Delta_1 \vdash p_1 \Leftarrow A_1 \qquad \Delta_2 \vdash p_2 \Leftarrow A_2}{\Delta_1, \Delta_2 \vdash (p_1, p_2) \Leftarrow A_1 \times A_2}$$

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

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

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

Fibonacci Example Program

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

$$\Sigma(\mathsf{fib}) \quad = \nu X \{\mathsf{head} : \mu Y \, \langle \mathsf{zero}\ 1 \mid \mathsf{suc}\ \ Y \rangle \,,\ \ \mathsf{tail} : X\}$$

$$\mathsf{Rules}(\mathsf{fib}) = \left\{ \begin{array}{lll} \cdot\ .\mathsf{head} & \mapsto & \mathsf{zero}\,() \\ \cdot\ .\mathsf{tail}\ .\mathsf{head} & \mapsto & \mathsf{suc}\,(\mathsf{zero}\,()) \\ \cdot\ .\mathsf{tail}\ .\mathsf{tail} & \mapsto & \mathsf{zipWith}\ \_{+}\_\ \ \mathsf{fib}\ (\mathsf{fib}\ .\mathsf{tail}) \end{array} \right\}$$

Evaluation

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

$$
\begin{array}{rcll}
E & ::= & \cdot & \text{Hole} \\
 & \mid & E\ e & \text{Application} \\
 & \mid & E.d & \text{Projection}
\end{array}
$$

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

(Co)pattern Matching

- $\boxed{t =^? p \searrow \sigma}$ Term t matches with pattern p under substitution σ.

$$\frac{}{t =^? x \searrow t/x} \qquad \frac{t =^? p \searrow \sigma}{c\ t =^? c\ p \searrow \sigma}$$

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

- $\boxed{E =^? q \searrow \sigma}$ Evaluation context E matches copattern q returning substitution σ.

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

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

Interactive Program Development

- Goal: cyclic stream of numbers.

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

- 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

- Last state:

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

- Split x in second clause.

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

- Fill right hand sides.

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

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

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

- Result splitting:

$$\frac{}{A \;\triangleleft|\; (\cdot \vdash \cdot \Rightarrow A)} \qquad \frac{A \;\triangleleft|\; \vec{Q}\;(\Delta \vdash q \Rightarrow B \to C)}{A \;\triangleleft|\; \vec{Q}\;(\Delta, x : B \vdash q\; x \Rightarrow C)}$$

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

- Variable splitting:

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

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

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.