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.
  \[
  \text{CoInductive } U : \text{Type} := \\
  | \text{inn} : U \rightarrow U.
  \]

  \[
  \text{CoFixpoint } u : U := \text{inn } u.
  \]

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

  \[
  \begin{align*}
  \text{case (inn } s) \text{ of inn } y \Rightarrow t & = t[s/y] \\
  \text{case (cofix } f) \text{ of inn } y \Rightarrow t & = \text{case } (f (\text{cofix } f)) \text{ of inn } y \Rightarrow t
  \end{align*}
  \]
A problem with subject reduction in Coq

\[
\begin{align*}
U : & \text{ Type } & \text{a codata type} \\
\text{inn} : & U \to U & \text{its (co)constructor} \\
u : & U & \text{inhabitant of } U \\
\text{u = cofix inn } & \text{ } & \text{u = inn(inn(...)} \\
\text{force} : & U \to U & \text{an identity} \\
\text{force} = & \lambda x. \text{case } x \text{ of inn } y \Rightarrow \text{inn } y \\
\text{eq} : & (x : U) \to x \equiv \text{force } x & \text{dep. elimination} \\
\text{eq} = & \lambda x. \text{case } x \text{ of inn } y \Rightarrow \text{refl} \\
\text{eq}_u : & u \equiv \text{inn } u & \text{offending term} \\
\text{eq}_u = & \text{eq } u \to \text{refl} & \not\exists \text{refl : } u \equiv \text{inn } u
\end{align*}
\]
Problematic: dependent matching on coinductive data.

\[ \Gamma \vdash u : U \quad \Gamma, \ y : U \vdash t : C(\text{inn} \ y) \]
\[ \Gamma \vdash \text{case } u \text{ of } \text{inn} \ y \Rightarrow t : C(u) \]

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].

\[ f : A \to B \quad a : A \]
\[ f \ a : B \]

- \( \lambda \)-abstraction is derived, secondary to application.
- Transfer this to other infinite objects: coinductive things.
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:

\[
\text{record } U : \text{Set where}
\]

\[
\text{coinductive}
\]

\[
\text{field}
\]

\[
\text{out : } U
\]

\[
\text{out : } U \rightarrow U \text{ is the experiment we can make on } U.
\]

Objects of type \( U \) are defined by the result of this experiment.
Infinite Objects Defined by Observation

- Defining a cofixpoint.
  \[ u : U \]
  \[ \text{out} \ u = u \]

- Defining the “constructor”.
  \[ \text{inn} : U \to U \]
  \[ \text{out} \ (\text{inn} \ x) = x \]

- We call \( \text{out} \ _ \) a projection copattern.
- And \( \_ \ x \) an application copattern.
- The whole thing \( \text{out} \ (\_ \ x) \) is a composite copattern.
Category-theoretic Perspective

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

\[
\begin{array}{c}
A \xrightarrow{s} F(A) \\
\downarrow \quad \downarrow \\
\nu F \xrightarrow{\text{out}} F(\nu F)
\end{array}
\]

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

\[\text{out}(\text{coit}(s)(a)) = F(\text{coit}(s))(s(a))\]
Instance: $U$

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

```
\begin{tikzpicture}
    \node (A) at (0,0) {$A$};
    \node (A1) at (0,-1) {$U$};
    \node (A2) at (2,0) {$A$};
    \node (U) at (2,-1) {$U$};
    \draw[->] (A) to node [midway, above] {$\text{id}$} (A1);
    \draw[->] (A) to node [midway, below] {$\text{coit}(id)$} (A2);
    \draw[->] (A1) to node [midway, left] {$\text{coit}(id)$} (A2);
    \draw[->] (A1) to node [midway, right] {$\text{out}$} (U);
    \draw[->] (A2) to node [midway, above] {$\text{coit}(id)$} (U);
\end{tikzpicture}
```

- Computation $\text{out}(u) = u$. 
**Instance: Colists of Natural Numbers**

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

\[
\begin{array}{c}
\mathbb{N} \xrightarrow{s} 1 + \mathbb{N} \times \mathbb{N} \\
\text{coit}(s) \downarrow \quad F(\text{coit}(s))
\end{array}
\]

\[
\begin{array}{c}
\text{Colist}(\mathbb{N}) \xrightarrow{\text{out}} 1 + \mathbb{N} \times \text{Colist}(\mathbb{N})
\end{array}
\]
Colists in Agda

- Colists as record.
  
  ```agda
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.

  ```agda
  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:

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 2 & 3 & 5 & 8 & \ldots \\
1 & 1 & 2 & 3 & 5 & 8 & 13 & \ldots \\
1 & 2 & 3 & 5 & 8 & 13 & 21 & \ldots \\
\end{array}
\]

\[\text{(fib)}\]
\[\text{(tail fib)}\]
\[\text{tail (tail fib)}\]

- This directly leads to a definition by copatterns:

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

- Strongly normalizing definition of \text{fib}!
Fibonacci

- Definition with \textbf{cons} not strongly normalizing.
  \[
  \text{fib} = 0 :: 1 :: \text{zipWith } _+_- \text{ fib (tail fib)}
  \]

- Diverges under Coq’s reduction strategy:
  \[
  \text{tail fib}
  = \text{tail} (0 :: 1 :: \text{zipWith } _+_- \text{ fib (tail fib)})
  = 1 :: \text{zipWith } _+_- \text{ fib (tail fib)}
  = 1 :: \text{zipWith } _+_- \text{ fib (tail (0 :: 1 :: \text{zipWith } _+_- \text{ fib (tail fib)}))}
  = \ldots
  \]
Type-theoretic motivation

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

<table>
<thead>
<tr>
<th>polarity</th>
<th>positive</th>
<th>negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear types</td>
<td>$1, \oplus, \otimes, \mu$</td>
<td>$\to, &amp;, \nu$</td>
</tr>
<tr>
<td>Agda types</td>
<td>data</td>
<td>$\to$, record</td>
</tr>
<tr>
<td>extension</td>
<td>finite</td>
<td>infinite</td>
</tr>
<tr>
<td>introduction</td>
<td>constructors</td>
<td>definition by copatterns</td>
</tr>
<tr>
<td>elimination</td>
<td>pattern matching</td>
<td>message passing</td>
</tr>
<tr>
<td>categorical</td>
<td>algebra</td>
<td>coalgebra</td>
</tr>
</tbody>
</table>
Types

\[ A, B, C ::= X \]
\[ \quad | P \quad | N \]

**Type variable**
**Positive type**
**Negative type**

\[ P ::= 1 \]
\[ \quad | A \times B \quad | \mu XD \]

**Unit type**
**Cartesian product**
**Data type**

\[ N ::= A \to B \]
\[ \quad | \nu XR \]

**Function type**
**Record type**

\[ D ::= \langle c_1 A_1 \mid \cdots \mid c_n A_n \rangle \]
\[ R ::= \{ d_1 : A_1, \ldots, d_n : A_n \} \]

**Variant (labeled sum)**
**Record (labeled product)**
Type examples

- **Data types (algebraic types):**

  \[
  \begin{align*}
  \text{List } A &= \mu X \langle \text{nil} 1 \mid \text{cons} (A \times X) \rangle \\
  \text{Nat} &= \mu X \langle \text{zero} 1 \mid \text{suc} X \rangle \\
  \text{Maybe } A &= \mu_\_ \langle \text{nothing} 1 \mid \text{just} A \rangle \\
  0 &= \mu_\_ \langle \rangle \quad \text{(positive empty type)}
  \end{align*}
  \]

- **Record types (coalggebraic types):**

  \[
  \begin{align*}
  \text{Stream } A &= \nu X \{ \text{head} : A, \text{tail} : X \} \\
  \text{Colist } A &= \nu X \{ \text{out} : \mu_\_ \langle \text{nil} 1 \mid \text{cons} (A \times X) \rangle \} \\
  \text{Vector } A &= \nu_\_ \{ \text{length} : \text{Nat}, \text{elems} : \text{List } A \} \\
  \top &= \nu_\_ \{ \} \quad \text{(negative unit type)}
  \end{align*}
  \]
Terms

\[ e, t, u ::= f \quad \text{Defined symbol (e.g. function)} \]
\[ x \quad \text{Variable} \]
\[ () \quad \text{Unit (empty tuple)} \]
\[ (t_1, t_2) \quad \text{Pair} \]
\[ c.t \quad \text{Constructor application} \]
\[ t_1 \ t_2 \quad \text{Application} \]
\[ t.d \quad \text{Destructor application} \]
Bidirectional Type Checking

- \( \Delta \vdash t \Rightarrow A \) In context \( \Delta \), the type of term \( t \) is inferred as \( A \).
  
  \[
  \begin{align*}
  \Delta \vdash f \Rightarrow \Sigma(f) \\
  \Delta \vdash x \Rightarrow A \\
  \Delta \vdash t \Rightarrow \nu X R \\
  \Delta(x) = A \\
  \Delta \vdash t \Rightarrow \nu X R
  \end{align*}
  \]

  \[
  \Delta \vdash t_1 \Rightarrow A_1 \rightarrow A_2 \\
  \Delta \vdash t_2 \Leftarrow A_1 \\
  \Delta \vdash t_1 \ t_2 \Rightarrow A_2
  \]

- \( \Delta \vdash t \Leftarrow A \) In context \( \Delta \), term \( t \) checks against type \( A \).
  
  \[
  \begin{align*}
  \Delta \vdash t \Rightarrow A \\
  A = C \\
  \Delta \vdash t \Leftarrow C \\
  \Delta \vdash t \Leftarrow D_c[\mu X D/X] \\
  \Delta \vdash c \ t \Leftarrow \mu X D \\
  \Delta \vdash (t_1, t_2) \Leftarrow A_1 \times A_2
  \end{align*}
  \]
Patterns and Copatterns

Patterns

\[ p ::= x \quad \text{Variable pattern} \]
\[ (\) \quad \text{Unit pattern} \]
\[ (p_1, p_2) \quad \text{Pair pattern} \]
\[ c \ p \quad \text{Constructor pattern} \]

Copatterns

\[ q ::= \_ \quad \text{Hole} \]
\[ q \ p \quad \text{Application copattern} \]
\[ q.d \quad \text{Destructor copattern} \]
Pattern Type Checking

- $\Delta \vdash p \leftarrow A$ Pattern $p$ checks against type $A$, yielding $\Delta$.

\[\begin{align*}
\Delta \vdash p & \leftarrow A \\
\Delta & \vdash D_c[\mu XD/X] \\
\Delta & \vdash c \; p \leftarrow \mu XD \\
\Delta_1 & \vdash p_1 \leftarrow A_1 \\
\Delta_2 & \vdash p_2 \leftarrow A_2 \\
\Delta_1, \Delta_2 & \vdash (p_1, p_2) \leftarrow A_1 \times A_2
\end{align*}\]

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

\[\begin{align*}
\Delta \mid A \vdash q & \Rightarrow \nu XR \\
\cdot & \mid A \vdash \cdot \Rightarrow A \\
\Delta & \mid A \vdash q.d \Rightarrow R_d[\nu XR/X] \\
\Delta_1 & \mid A \vdash q \Rightarrow B \Rightarrow C \\
\Delta_2 & \vdash p \leftarrow B \\
\Delta_1, \Delta_2 & \mid A \vdash q \; p \Rightarrow C
\end{align*}\]
Fibonacci Example Program

- Program consists of type signatures $\Sigma$ and rewrite rules $\text{Rules}$.
- Example entries for $\text{fib}$.

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

$$\text{Rules}(\text{fib}) = \begin{cases} 
  \cdot \text{.head} & \mapsto \text{zero } () \\
  \cdot \text{.tail .head} & \mapsto \text{suc } (\text{zero } ()) \\
  \cdot \text{.tail .tail} & \mapsto \text{zipWith } _+\_ \ \text{fib } (\text{fib .tail}) 
\end{cases}$$
Evaluation

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

$$E ::= \cdot \quad \text{Hole}$$
$$\quad | \quad E \ e \quad \text{Application}$$
$$\quad | \quad E.d \quad \text{Projection}$$

- To reduce redex, we need to match $E$ against copatterns $q$. 
(Co)pattern Matching

- **Term** $t = ? p \sigma$  Term $t$ matches with pattern $p$ under substitution $\sigma$.

  \[
  \begin{align*}
  t = ? x \sigma & \quad \text{for } t / x \\
  c \ t = ? c \ p \sigma & \quad \text{for } c \ t = ? c \ p \sigma
  \end{align*}
  \]

- **Evaluation context** $E = ? q \sigma$  Evaluation context $E$ matches copattern $q$ returning substitution $\sigma$.

  \[
  \begin{align*}
  \cdot = ? \cdot \sigma & \quad \text{for } E. d = ? q. d \sigma \\
  E = ? q \sigma & \quad \text{for } E \ t = ? q \ p \sigma' \\
  \end{align*}
  \]
Goal: cyclic stream of numbers.

\[
\text{cycleNats} : \mathbb{N} \to \text{Stream } \mathbb{N} \\
\text{cycleNats } n = n, n - 1, \ldots, 1, 0, N, N - 1, \ldots, 1, 0, \ldots
\]

Fictuous interactive Agda session.

\[
\text{cycleNats} : \text{Nat} \to \text{Stream } \text{Nat} \\
\text{cycleNats } = ?
\]

Split result (function).

\[
\text{cycleNats } x = ?
\]

Split result again (stream).

\[
\text{head } (\text{cycleNats } x) = ? \\
\text{tail } (\text{cycleNats } x) = ?
\]
Interactive Program Development

- Last state:
  \[ \text{head} \ (\text{cycleNats} \ x) = ? \]
  \[ \text{tail} \ (\text{cycleNats} \ x) = ? \]

- Split \( x \) in second clause.
  \[ \text{head} \ (\text{cycleNats} \ x) = ? \]
  \[ \text{tail} \ (\text{cycleNats} \ 0) = ? \]
  \[ \text{tail} \ (\text{cycleNats} \ (1 + x')) = ? \]

- Fill right hand sides.
  \[ \text{head} \ (\text{cycleNats} \ x) = x \]
  \[ \text{tail} \ (\text{cycleNats} \ 0) = \text{cycleNats} \ N \]
  \[ \text{tail} \ (\text{cycleNats} \ (1 + x')) = \text{cycleNats} \ x' \]
Coverage algorithm:

Start with the trivial covering \((\text{copattern} \cdot \text{“hole”})\).

Repeat

- split result or
- split a pattern variable

until computed covering matches user-given patterns.
Coverage Rules

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

- **Result splitting:**
  
  \[
  A \triangleright | (\cdot \vdash \cdot \Rightarrow A) \quad A \triangleright | \vec{Q} (\Delta \vdash q \Rightarrow B \Rightarrow C) \quad A \triangleright | \vec{Q} (\Delta, x : B \vdash q \; x \Rightarrow C) 
  \]

  \[
  A \triangleright | \vec{Q} (\Delta \vdash q \Rightarrow \nuXR) 
  \]

  \[
  A \triangleright | \vec{Q} (\Delta \vdash q \cdot d \Rightarrow R_d[\nuXR/X])_{d \in R} 
  \]

- **Variable splitting:**
  
  \[
  A \triangleright | \vec{Q} (\Delta, x : A_1 \times A_2 \vdash q \Rightarrow C) \quad A \triangleright | \vec{Q} (\Delta, x_1 : A_1, x_2 : A_2 \vdash q[(x_1, x_2)/x] \Rightarrow C) 
  \]

  \[
  A \triangleright | \vec{Q} (\Delta, x : \muXD \vdash q \Rightarrow C) 
  \]

  \[
  A \triangleright | \vec{Q} (\Delta, x' : D_c[\muXD/X] \vdash q[c \; x'/x] \Rightarrow C)_{c \in D} 
  \]
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:
  
  \textit{Abel, Pientka, Thibodeau, and Setzer}
  
  \textit{Copatterns – Programming Infinite Structures by Observation.}

- Related Work:
  
  - Cockett et al. (1990s): Charity.