Copatterns
Programming Infinite Objects by Observations

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich

Mathematical Logic Seminar
Ludwig-Maximilians-University Munich
7 November 2012
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.
  
  ```coq
  CoInductive U : Type :=
  | inn : U -> U.
  
  CoFixpoint u : U := inn u.
  ```

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

  ```coq
  case (inn s) of inn y \Rightarrow t = t[s/y] 
  case (cofix f) of inn y \Rightarrow t = case (f (cofix f)) of inn y \Rightarrow t 
  ```
A problem with subject reduction in Coq

\[ U : \text{Type} \quad \text{a codata type} \]
\[ \text{inn} : U \rightarrow U \quad \text{its (co)constructor} \]
\[ u : U \quad \text{inhabitant of } U \]
\[ u = \text{cofix inn} \quad u = \text{inn(inn(…)} \]
\[ \text{force} : U \rightarrow U \quad \text{an identity} \]
\[ \text{force} = \lambda x. \text{case } x \text{ of inn } y \Rightarrow \text{inn } y \]
\[ \text{eq} : (x : U) \rightarrow x \equiv \text{force } x \quad \text{dep. elimination} \]
\[ \text{eq} = \lambda x. \text{case } x \text{ of inn } y \Rightarrow \text{refl} \]
\[ \text{eq}_u : u \equiv \text{inn } u \quad \text{offending term} \]
\[ \text{eq}_u = \text{eq } u \rightarrow \text{refl} \quad \not\vdash \text{refl : } u \equiv \text{inn } u \]
Analysis

- 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].

\[
\begin{align*}
f &: A \rightarrow B \\
  a &: A \\
\hline
  f \ a &: B
\end{align*}
\]

• \(\lambda\)-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:
  
  ```agda
  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.
  ```
Introduction

Infinite Objects Defined by Observation

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

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

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

\[ A \xrightarrow{s} F(A) \]

\[ \nu F \xrightarrow{\text{out}} F(\nu F) \]

- 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 $u = \text{coit}(\text{id})(a)$ for arbitrary $a : A$.

\[
\begin{array}{ccc}
A & \xrightarrow{\text{id}} & A \\
\downarrow & & \downarrow \\
\text{coit}(\text{id}) & & \text{coit}(\text{id}) \\
\downarrow & & \downarrow \\
U & \xrightarrow{\text{out}} & U \\
\end{array}
\]

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

Andreas Abel (LMU) Agda Logik 10 / 32
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
    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\} \rightarrow (A \rightarrow B \rightarrow C) \rightarrow
  Stream A \rightarrow Stream B \rightarrow 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:

\[
\begin{array}{ccccccccc}
0 & 1 & 1 & 2 & 3 & 5 & 8 & \ldots & (\text{fib}) \\
1 & 1 & 2 & 3 & 5 & 8 & 13 & \ldots & (\text{tail fib}) \\
1 & 2 & 3 & 5 & 8 & 13 & 21 & \ldots & \text{tail (tail fib)} \\
\end{array}
\]

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

- Diverges under Coq’s reduction strategy:
  \[
  \text{tail} \ \text{fib} \\
  = \text{tail} \ (0 :: 1 :: \text{zipWith} \ _+_ \ \text{fib} \ (\text{tail} \ \text{fib})) \\
  = 1 :: \text{zipWith} \ _+_ \ \text{fib} \ (\text{tail} \ \text{fib}) \\
  = 1 :: \text{zipWith} \ _+_ \ \text{fib} \\
  \quad (\text{tail} \ (0 :: 1 :: \text{zipWith} \ _+_ \ \text{fib} \ (\text{tail} \ \text{fib}))) \\
  = ... 
  \]
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>$\rightarrow, &amp; , \nu$</td>
</tr>
<tr>
<td>Agda types</td>
<td>data</td>
<td>$\rightarrow, \text{record}$</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

$A, B, C ::= X$

$P ::= 1$
$A \times B$
$\mu XD$

$N ::= A \to B$
$\nu XR$

$D ::= \langle c_1 A_1 | \cdots | c_n A_n \rangle$

$R ::= \{ d_1 : A_1, \ldots, d_n : A_n \}$
Type examples

- **Data types (algebraic types):**
  
  List $A = \mu X \langle \text{nil} \ 1 \ | \ \text{cons} \ (A \times X) \rangle$
  
  Nat = $\mu X \langle \text{zero} \ 1 \ | \ \text{suc} \ X \rangle$
  
  Maybe $A = \mu_{-} \langle \text{nothing} \ 1 \ | \ \text{just} \ A \rangle$
  
  0 = $\mu_{-} \langle \rangle$ (positive empty type)

- **Record types (coalgebraic types):**
  
  Stream $A = \nu X \{ \text{head} : A, \ \text{tail} : X \}$
  
  Colist $A = \nu X \{ \text{out} : \mu_{-} \langle \text{nil} \ 1 \ | \ \text{cons} \ (A \times X) \rangle \}$
  
  Vector $A = \nu_{-}\{ \text{length} : \text{Nat}, \ \text{elems} : \text{List} A \}$
  
  $\top = \nu_{-}\{ \}$ (negative unit type)
Syntax

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

**Δ ⊢ t ⇒ A**  In context Δ, the type of term t is inferred as A.

- Δ ⊢ f ⇒ Σ(f)  
- Δ ⊢ x ⇒ A  
- Δ ⊢ t ⇒ νXR  
- Δ ⊢ t.d ⇒ R_d[νXR/X]  

- Δ ⊢ t₁ ⇒ A₁ → A₂  
- Δ ⊢ t₂ ⇐ A₁  
- Δ ⊢ t₁ t₂ ⇒ A₂

**Δ ⊢ t ⇐ A**  In context Δ, term t checks against type A.

- Δ ⊢ t ⇒ A  
- A = C  
- Δ ⊢ t ⇐ D_c[µXD/X]  
- Δ ⊢ c t ⇐ µXD

- Δ ⊢ () ⇐ 1  
- Δ ⊢ (t₁, t₂) ⇐ A₁ × A₂
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**

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

- \( \Delta \vdash p \Leftarrow A \)  
  Pattern \( p \) checks against type \( A \), yielding \( \Delta \).

\[
\begin{align*}
  x : A & \vdash x \Leftarrow A \\
  \Delta & \vdash c \ p \Leftarrow \mu XD
\end{align*}
\]

\[
\begin{align*}
  \Delta_1 & \vdash p_1 \Leftarrow A_1 & \Delta_2 & \vdash p_2 \Leftarrow A_2 \\
  \Delta_1, \Delta_2 & \vdash (p_1, p_2) \Leftarrow A_1 \times A_2
\end{align*}
\]

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

\[
\begin{align*}
  \Delta & \mid A \vdash q \Rightarrow \nu XR \\
  \cdot & \mid A \vdash \cdot \Rightarrow A & \Delta & \mid A \vdash q.d \Rightarrow R_d[\nu XR/X]
\end{align*}
\]

\[
\begin{align*}
  \Delta_1 & \mid A \vdash q \Rightarrow B \rightarrow C & \Delta_2 & \vdash p \Leftarrow B \\
  \Delta_1, \Delta_2 & \mid A \vdash q \ p \Rightarrow C
\end{align*}
\]
Fibonacci Example Program

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

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

$$\text{Rules}(\text{fib}) = \left\{ \begin{array}{l}
\cdot \text{.head} \mapsto \text{zero} () \\
\cdot \text{.tail .head} \mapsto \text{suc} (\text{zero} ()) \\
\cdot \text{.tail .tail} \mapsto \text{zipWith} _+ \ _\text{fib} (\text{fib} .\text{tail}) 
\end{array} \right\}$$
Evaluation

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

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

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

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

- $t =? x \sigma$ $t$ matches with pattern $x$.
- $c t =? c p \sigma$ Substitution $\sigma$ is applied.

- $t_1 =? p_1 \sigma_1$ $t_2 =? p_2 \sigma_2$ $t_1, t_2$ match with $p_1, p_2$.

- $(t_1, t_2) =? (p_1, p_2) \sigma_{1,2}$ Tuple matches with tuple.

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

- $E =? q \sigma$ $E$ matches copattern $q$.
- $E.d =? q.d \sigma$ Tuple $d$ matches with $q.d$.

- $E t =? q p \sigma, \sigma'$ $t$ matches with $p$ after substitution $\sigma$, and $E$ matches with $q$ after substitution $\sigma'$.
Interactive Program Development

- Goal: cyclic stream of numbers.

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

- Fictuous interactive Agda session.

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

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

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

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

\( A \iff \vec{Q} \) Typed copatterns \( \vec{Q} \) cover elimination of type \( A \).

- **Result splitting:**
  \[
  \frac{A \iff (\cdot \vdash \cdot \Rightarrow A)}{A \iff \vec{Q}}
  \]
  \[
  \frac{A \iff \vec{Q} \ (\Delta \vdash q \Rightarrow B \Rightarrow C)}{A \iff \vec{Q} \ (\Delta, x : B \vdash q \ x \Rightarrow C)}
  \]
  \[
  \frac{A \iff \vec{Q} \ (\Delta \vdash q \Rightarrow \nu XR)}{A \iff \vec{Q} \ (\Delta \vdash q \ . d \Rightarrow R_d[\nu XR/X])_{d \in R}}
  \]

- **Variable splitting:**
  \[
  \frac{A \iff \vec{Q} \ (\Delta, x : A_1 \times A_2 \vdash q \Rightarrow C)}{A \iff \vec{Q} \ (\Delta, x_1 : A_1, x_2 : A_2 \vdash q[(x_1, x_2)/x] \Rightarrow C)}
  \]
  \[
  \frac{A \iff \vec{Q} \ (\Delta, x : \mu XD \vdash q \Rightarrow C)}{A \iff \vec{Q} \ (\Delta, x' : D_c[\mu XD/X] \vdash q[c \ x'/x] \Rightarrow C)_{c \in D}}
  \]
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.