Copatterns
Programming Infinite Objects by Observations

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich, Germany

Institute of Cybernetics
Tallinn, Estonia
4 July 2013
Crash course “Programming in the Infinite”

Final Exam
Problem 1 (Duality): Complete this table!

<table>
<thead>
<tr>
<th>finite</th>
<th>infinite</th>
</tr>
</thead>
<tbody>
<tr>
<td>algebra</td>
<td>coalgebra</td>
</tr>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>constructors</td>
<td>destructors</td>
</tr>
<tr>
<td>pattern matching</td>
<td></td>
</tr>
</tbody>
</table>
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.
   - 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.

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

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

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

```coq
  case cons a s of cons x y ⇒ t = t[a/x][s/y]
  case cofix f of branches = case f (cofix f) of branches
```

- Leads to loss of subject reduction. [Gimenez, 1996; Oury, 2008]
Issue 1: Loss of Subject Reduction

Stream : Type
cons : \( \mathbb{N} \to \text{Stream} \to \text{Stream} \)
zeros : Stream
zeros = cofix (cons 0)
force : Stream \to \text{Stream}
force s = case s of cons x y => cons x y

eq : (s : Stream) \to s \equiv \text{force s}
eq s = case s of cons x y => refl
eqzeros : zeros \equiv cons 0 zeros
\n\n\n\n
Analysis

- Problematic: dependent matching on coinductive data.
  \[
  \Gamma \vdash s : \text{Stream} \quad \Gamma, \ x : \mathbb{N}, \ y : \text{Stream} \vdash t : C(\text{cons} \times y)
  \]
  \[
  \Gamma \vdash \text{case } s \text{ of } \text{cons} \times y \Rightarrow t : C(s)
  \]

- [McBride, 2009]: Let’s see how things unfold.
Issue 2: Deep Guardedness Not Supported

- Fibonacci sequence obeys recurrence:

\[
\begin{array}{c|cccccccc}
\text{zipWith} (\_\text{+_}\_ ) & 0 & 1 & 1 & 2 & 3 & 5 & 8 & \ldots \\
0 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & \ldots
\end{array}
\]

- Direct recursive definition:

\[
\begin{align*}
\text{fib} &= \text{cons } 0 \ (\text{cons } 1 \ (\text{zipWith} \ _\text{+_}\_ \ \text{fib} \ (\text{tail} \ \text{fib}))) \\
\text{fib} &= \text{cons } 0 \ (F \ (\text{tail} \ \text{fib}))
\end{align*}
\]

- Diverges under Coq’s reduction strategy:

\[
\begin{align*}
\text{tail} \ \text{fib} &= F \ (\text{tail} \ \text{fib}) \\
&= F \ (F \ (\text{tail} \ \text{fib})) \\
&= \ldots
\end{align*}
\]
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.
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].

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

- \(\lambda\)-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.

\[
\begin{align*}
\text{record} & \quad \text{Stream} : \text{Set} \quad \text{where} \\
\quad & \quad \text{coinductive} \\
\quad & \quad \text{field} \\
\quad & \quad \quad \text{head} : \mathbb{N} \\
\quad & \quad \quad \text{tail} : \text{Stream}
\end{align*}
\]

- **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.
  
  \[
  \text{zeros} : \text{Stream} \\
  \text{head} \ \text{zeros} = 0 \\
  \text{tail} \ \text{zeros} = \text{zeros}
  \]

- Defining the “constructor”.
  
  \[
  \text{cons} : \mathbb{N} \rightarrow \text{Stream} \rightarrow \text{Stream} \\
  \text{head} ((\text{cons} \ x) \ y) = x \\
  \text{tail} ((\text{cons} \ x) \ y) = y
  \]

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

\[ q ::= \cdot \quad \text{Hole} \]
\[ | q \ p \quad \text{Application copattern} \]
\[ | d \ q \quad \text{Projection/destructor copattern} \]

Definitions

\[ q_1[f/\cdot] = t_1 \]
\[ \vdots \]
\[ q_n[f/\cdot] = t_n \]
Definition by Observation

Coalgebras

Category-theoretic Perspective

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

\[
\begin{align*}
A & \xrightarrow{s} F(A) \\
\nu F & \xrightarrow{\text{force}} F(\nu F)
\end{align*}
\]

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

\[
\text{force}(\text{coit}(s)(a)) = F(\text{coit}(s))(s(a))
\]
Instance: Stream

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

Computation: $\text{(head, tail)}(\text{coit}(s)()) = (0, \text{coit}(s)())$.
Deep Copatterns: Fibonacci-Stream

- Fibonacci sequence obeys this recurrence:

\[
\begin{array}{c|cccccccc}
\text{zipWith } (+) & 0 & 1 & 1 & 2 & 3 & 5 & 8 & \ldots \\

tail (fib) & 1 & 1 & 2 & 3 & 5 & 8 & 13 & \ldots \\
\end{array}
\]

This directly leads to a definition by copatterns:

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

- Strongly normalizing definition of fib!
Type-Based Termination

- Termination by recursion on smaller size (wellfounded induction).

\[
i : \text{Size}, \quad f : \forall j < i. \ \text{Nat}^i \rightarrow C \vdash t : \text{Nat}^i \rightarrow C
\]
\[
\vdash \text{fix } f \cdot t : \forall i. \ \text{Nat}^i \rightarrow C
\]

- Shift of perspective: from size of argument to depth of observation on function.

\[
i : \text{Size}, \quad f : \forall j < i. \ \text{A}^j \vdash t : \text{A}^i
\]
\[
\vdash \text{fix } f \cdot t : \forall i. \ \text{A}^i
\]

- Extend to observation on streams:

\[
i : \text{Size}, \quad f : \forall j < i. \ \text{Stream}^j \text{A} \vdash t : \text{Stream}^i \text{A}
\]
\[
\vdash \text{fix } f \cdot t : \forall i. \ \text{Stream}^i \text{A}
\]
Sized Streams

- Semantic idea: Inflationary greatest fixed-point.

\[ \nu^i F = \bigcap_{j<i} F(\nu^j F) \]

- Constructors/destructors:

\[ \nu^i F \xleftarrow{\text{out}} \xrightarrow{\text{inn}} \forall j<i. F(\nu^j F) \]

- Typing of projections:

\[
\begin{align*}
\text{s : Stream}^i A & \quad \Rightarrow \quad s.\text{head} : \forall j<i. A \\
\text{s : Stream}^i A & \quad \Rightarrow \quad s.\text{tail} : \forall j<i. \text{Stream}^i A
\end{align*}
\]
Type-Based Productivity of Fibonacci Stream

- Sized version of \( \text{zipWith} \).

\[
\text{zipWith} : \forall i \leq \infty. \ |i| \Rightarrow \forall A : *. \forall B : *. \forall C : *.
\]

\[
(A \rightarrow B \rightarrow C) \rightarrow
\]

\[
\text{Stream}^i A \rightarrow \text{Stream}^i B \rightarrow \text{Stream}^i C
\]

\[
\text{zipWith} \ i \ A \ B \ C \ f \ s \ t \ . \text{head} \ j = f \ (s \ . \text{head} \ j) \ (t \ . \text{head} \ j)
\]

\[
\text{zipWith} \ i \ A \ B \ C \ f \ s \ t \ . \text{tail} \ j = \text{zipWith} \ j \ A \ B \ C \ f
\]

\[
(s \ . \text{tail} \ j) \ (t \ . \text{tail} \ j)
\]

- Productivity of \( \text{fib} \).

\[
\text{fib} : \forall i. \ |i| \Rightarrow \text{Stream}^i \mathbb{N}
\]

\[
\text{fib} \ i \ . \text{head} \ j = 0
\]

\[
\text{fib} \ i \ . \text{tail} \ j \ . \text{head} \ k = 1
\]

\[
\text{fib} \ i \ . \text{tail} \ j \ . \text{tail} \ k = \text{zipWith} \ k \ \mathbb{N} \ \mathbb{N} \ \mathbb{N} \ (+) \ (\text{fib} \ k) \ (\text{fib} \ j \ . \text{tail} \ k)
\]
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.

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

- Split result (function).

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

- Split result again (stream).

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

- Finish first clause:

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

- Split \( x \) in second clause.

  \[
  \text{head \ (cycleNats \ x) \ = \ x} \\
  \text{tail \ (cycleNats \ 0) \ = \ ?} \\
  \text{tail \ (cycleNats \ (1 + x')) \ = \ ?}
  \]

- Fill remaining right hand sides.

  \[
  \text{head \ (cycleNats \ x) \ = \ x} \\
  \text{tail \ (cycleNats \ 0) \ = \ cycleNats \ N} \\
  \text{tail \ (cycleNats \ (1 + x')) \ = \ cycleNats \ x'}
  \]
Coverage

- Coverage algorithm:
- Start with the trivial covering.
- Repeat
  - split a pattern variable
  until computed covering matches user-given patterns.
Coverage

Copattern Coverage

- Coverage algorithm:
  - Start with the trivial covering. (Copattern \( \cdot \) “hole”)
  - Repeat
    - split result or
    - split a pattern variable
  until computed covering matches user-given patterns.
## Deriving Covering Set of Clauses

<table>
<thead>
<tr>
<th>Start</th>
<th>( \vdash \cdot : \mathbb{N} \rightarrow \text{Stream} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split function</td>
<td>( \vdash x : \mathbb{N} \rightarrow \cdot x : \text{Stream} )</td>
</tr>
<tr>
<td>Split stream</td>
<td>( \vdash x : \mathbb{N} \rightarrow \text{head} (\cdot x) : \mathbb{N} ) ( \vdash x : \mathbb{N} \rightarrow \text{tail} (\cdot x) : \text{Stream} )</td>
</tr>
<tr>
<td>Split var.</td>
<td>( \vdash x : \mathbb{N} \rightarrow \text{head} (\cdot x) : \mathbb{N} ) ( \vdash \text{tail} (\cdot 0) : \text{Stream} ) ( \vdash x' : \mathbb{N} \rightarrow \text{tail} (\cdot (1 + x')) : \text{Stream} )</td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>finite / positive / type checking</th>
<th>finite / positive / type checking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tuple</strong></td>
<td><strong>data</strong></td>
</tr>
<tr>
<td>$A_1 \times A_2$</td>
<td>$\mu,+$</td>
</tr>
<tr>
<td>$(t_1, t_2)$</td>
<td>$c \ t$</td>
</tr>
<tr>
<td>$(p_1, p_2)$</td>
<td>$c \ p$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>infinite / negative / type inference</th>
<th>infinite / negative / type inference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>function</strong></td>
<td><strong>record</strong></td>
</tr>
<tr>
<td>$A_1 \rightarrow A_2$</td>
<td>$\nu, &amp;$</td>
</tr>
<tr>
<td>$q \ p$</td>
<td>$d \ q$</td>
</tr>
<tr>
<td>$e \ t$</td>
<td>$d \ e$</td>
</tr>
</tbody>
</table>
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!

```haskell
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'
```

Andreas Abel ()

Copatterns

IOC 2013 25 / 33
Conclusions

Future work:
- MiniAgda: A productivity checker with sized types.
- Prove strong normalization.
- TODO: Integrate copatterns into Agda’s kernel.

Related Work:
- Cockett et al. (1990s): Charity.
Crash course “Programming in the Infinite”

Model Solution

Problem 1 (Duality): Complete this table!

<table>
<thead>
<tr>
<th>finite</th>
<th>infinite</th>
</tr>
</thead>
<tbody>
<tr>
<td>algebra</td>
<td>coalgebra</td>
</tr>
<tr>
<td>inductive</td>
<td>coinductive</td>
</tr>
<tr>
<td>constructors</td>
<td>destructors</td>
</tr>
<tr>
<td>pattern matching</td>
<td>copattern matching</td>
</tr>
</tbody>
</table>
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 $\text{coit}(s)(n) = (n, n + 1, n + 2, \ldots)$. 

```
\[ \begin{array}{c}
\mathbb{N} & \xrightarrow{s} & 1 + \mathbb{N} \times \mathbb{N} \\
\text{coit}(s) \downarrow & & \downarrow F(\text{coit}(s)) \\
\text{Colist}(\mathbb{N}) & \xrightarrow{\text{force}} & 1 + \mathbb{N} \times \text{Colist}(\mathbb{N}) \\
\end{array} \]
```
Colists in Agda

- Colists as record.

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

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

- Sequence of natural numbers.

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

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

- **Result splitting:**
  
  \[
  \begin{align*}
  A & \triangleright \bar{Q} \\
  \begin{array}{c}
  A \triangleright (\vdash \cdot : A) \\
  \vdash \cdot : A
  \end{array} & \quad \begin{array}{c}
  \ldots (\Delta \vdash q : B \to C) \\
  \vdash \cdot : A
  \end{array} \\
  \vdash \cdot : A & \quad \begin{array}{c}
  \ldots (\Delta, x : B \vdash q \times : C) \\
  \vdash \cdot : A
  \end{array} \\
  \vdash \cdot : A & \quad \begin{array}{c}
  \ldots (\Delta \vdash q : R) \\
  \vdash \cdot : A
  \end{array} \\
  \vdash \cdot : A & \quad \begin{array}{c}
  \ldots (\Delta \vdash d q : R_d)_{d \in R}
  \end{array}
  \end{align*}
  \]

- **Variable splitting:**
  
  \[
  \begin{align*}
  \ldots (\Delta, x : A_1 \times A_2 \vdash q[x] : C) & \\
  \ldots (\Delta, x_1 : A_1, x_2 : A_2 \vdash q[(x_1, x_2)] : C) & \\
  \ldots (\Delta, x : D \vdash q[x] : C) & \\
  \ldots (\Delta, x' : D_c \vdash q[c x'] : C)_{c \in D}
  \end{align*}
  \]
Type-theoretic background

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></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>