Copatterns
Programming Infinite Objects by Observations

A. Abel\textsuperscript{1}  B. Pientka\textsuperscript{2}  D. Thibodeau\textsuperscript{2}  A. Setzer\textsuperscript{3}

\textsuperscript{1}Department of Computer Science
Ludwig-Maximilians-University Munich, Germany

\textsuperscript{2}School of Computer Science
McGill University, Montreal, Canada

\textsuperscript{3}Computer Science
Swansea University, Wales, UK

Principles of Programming Languages
Roma, Italy
23 January 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>
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.
  
  \[
  \text{CoInductive} \quad \text{Stream} : \text{Type} := \\
  \mid \text{cons} \ (\text{head} : \text{nat}) \ (\text{tail} : \text{Stream}).
  \]

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

  \[
  \begin{align*}
  \text{case} \ \text{cons} \ a \ s \ \text{of} \ \text{cons} \ x \ y \ \Rightarrow \ t &= t[a/x][s/y] \\
  \text{case} \ \text{cofix} \ f \ \text{of} \ \text{branches} &= \text{case} \ f \ (\text{cofix} \ f) \ \text{of} \ \text{branches}
  \end{align*}
  \]

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

Stream : Type
cons : \( \mathbb{N} \rightarrow \text{Stream} \rightarrow \text{Stream} \)

zeros : Stream
zeros = cofix (cons 0)

force : Stream \rightarrow \text{Stream}
force s = case s of cons x y \Rightarrow cons x y

eq : (s : \text{Stream}) \rightarrow s \equiv \text{force } s
eq s = case s of cons x y \Rightarrow \text{refl}

eq_{zeros} : \text{zeros } \equiv \text{cons 0 zeros}
eq_{zeros} = \text{eq zeros } \rightarrow \text{refl}
Analysis

Problematic: dependent matching on coinductive data.

\[
\Gamma \vdash s : \text{Stream} \quad \Gamma, x : \mathbb{N}, y : \text{Stream} \vdash t : C(\text{cons } x \ y)
\]

\[
\Gamma \vdash \text{case } s \text{ of } \text{cons } x \ y \Rightarrow t : C(s)
\]

[McBride, 2009]: Let’s see how things unfold.
Fibonacci sequence obeys recurrence:

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

Direct recursive definition:

\[
\text{fib} = \text{cons } 0 \ (\text{cons } 1 \ (\text{zipWith } (+\_ ) \ \text{fib} \ (\text{tail fib})))
\]

\[
\text{fib} = \text{cons } 0 \ (\ F \ (\text{tail fib}))
\]

Diverges under Coq’s reduction strategy:

\[
\text{tail fib}
\]

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

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

\[
= \ldots
\]
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. (In progress.)
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.
- Typical semantic view of functions.
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.

```haskell
record Stream : Set where
  coinductive
  field
    head : ℕ
    tail : Stream
```

- **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.
  
  ```
  zeros : Stream
  head zeros = 0
  tail zeros = zeros
  ```

- Defining the “constructor”.
  
  ```
  cons : N → Stream → Stream
  head ((cons x) y) = x
  tail ((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**
  \[ p ::= \begin{array}{l}
  x \quad \text{Variable pattern} \\
  () \quad \text{Unit pattern} \\
  (p_1, p_2) \quad \text{Pair pattern} \\
  c \ p \quad \text{Constructor pattern}
  \end{array} \]

- **Copatterns**
  \[ q ::= \begin{array}{l}
  \cdot \quad \text{Hole} \\
  q \ p \quad \text{Application copattern} \\
  d \ q \quad \text{Projection/destructor copattern}
  \end{array} \]

- **Definitions**
  \[ q_1[f/\cdot] = t_1 \]
  \[ \vdots \]
  \[ q_n[f/\cdot] = t_n \]
Category-theoretic Perspective

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

![Diagram]

- 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 $\text{zeros} = \text{coit}(s)()$.

\[
\begin{array}{c}
1 \xrightarrow{s} \mathbb{N} \times 1 \\
\downarrow \quad \downarrow \\
\text{coit}(s) \quad F(\text{coit}(s)) \quad \text{head, tail} \quad \downarrow \\
\text{Stream} \quad \rightarrow \quad \mathbb{N} \times \text{Stream}
\end{array}
\]

- Computation: $(\text{head}, \text{tail})(\text{coit}(s)()) = (0, \text{coit}(s)())$. 
Fibonacci sequence obeys this recurrence:

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

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

This directly leads to a definition by copatterns:

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

Strongly normalizing definition of \text{fib}!
Interactive Program Development

- Goal: cyclic stream of numbers.
  
  \[
  \text{cycleNats} \quad : \quad \mathbb{N} \rightarrow \text{Stream} \, \mathbb{N} \\
  \text{cycleNats} \, n \quad = \quad n, n - 1, \ldots, 1, 0, \, N, \, N - 1, \ldots, 1, 0, \ldots
  \]

- Fictuous interactive Agda session.

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

- Split result (function).

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

- Split result again (stream).

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

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

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

- Fill remaining 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

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

- 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.
## Deriving Covering Set of Clauses

<table>
<thead>
<tr>
<th>Role</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>($\vdash \cdot : \mathbb{N} \to \text{Stream}$)</td>
</tr>
<tr>
<td>split function</td>
<td>($x:\mathbb{N} \vdash \cdot \ x : \text{Stream}$)</td>
</tr>
<tr>
<td>split stream</td>
<td>($x:\mathbb{N} \vdash \text{head} (\cdot \ x) : \mathbb{N}$)</td>
</tr>
<tr>
<td></td>
<td>($x:\mathbb{N} \vdash \text{tail} (\cdot \ x) : \text{Stream}$)</td>
</tr>
<tr>
<td>split var.</td>
<td>($x:\mathbb{N} \vdash \text{head} (\cdot x) : \mathbb{N}$)</td>
</tr>
<tr>
<td></td>
<td>($\vdash \text{tail} (\cdot 0) : \text{Stream}$)</td>
</tr>
<tr>
<td></td>
<td>($x' : \mathbb{N} \vdash \text{tail} (\cdot (1 + x')) : \text{Stream}$)</td>
</tr>
</tbody>
</table>
## Syntax

<table>
<thead>
<tr>
<th>Finite / Positive / Type Checking</th>
<th>Type Introduction</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple</td>
<td>( A_1 \times A_2 )</td>
<td>((t_1, t_2))</td>
</tr>
<tr>
<td>Data</td>
<td>( \mu, + )</td>
<td>(ct)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Infinite / Negative / Type Inference</th>
<th>Type Copattern Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>( A_1 \to A_2 )</td>
</tr>
<tr>
<td>Record</td>
<td>( \nu, &amp; )</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’
```
Conclusions

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

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

\[
A \triangleright\!
\begin{array}{c}
\tilde{Q}
\end{array}
\]
Typed copatterns \(\tilde{Q}\) cover elimination of type \(A\).

- **Result splitting:**

\[
A \triangleright\! (\vdash \cdot : A) \quad \begin{array}{l}
\ldots (\Delta \vdash q : B \rightarrow C) \\
\ldots (\Delta, x : B \vdash q \times : C)
\end{array}
\]

\[
\begin{array}{c}
\ldots (\Delta \vdash q : R) \\
\ldots (\Delta \vdash d q : R_d)_{d \in R}
\end{array}
\]

- **Variable splitting:**

\[
\begin{array}{l}
\ldots (\Delta, x : A_1 \times A_2 \vdash q[x] : C)
\end{array}
\]

\[
\begin{array}{l}
\ldots (\Delta, x_1:A_1, x_2:A_2 \vdash q[(x_1, x_2)] : C)
\end{array}
\]

\[
\begin{array}{l}
\ldots (\Delta, x:D \vdash q[x] : C)
\end{array}
\]

\[
\begin{array}{l}
\ldots (\Delta, x':D_c \vdash q[c x'] : C)_{c \in D}
\end{array}
\]
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>$\rightarrow, &amp;, \nu$</td>
</tr>
<tr>
<td>Agda types</td>
<td>data</td>
<td>definition by copatterns</td>
</tr>
<tr>
<td>extension</td>
<td>finite</td>
<td>infinite</td>
</tr>
<tr>
<td>introduction</td>
<td>constructors</td>
<td>message passing</td>
</tr>
<tr>
<td>elimination</td>
<td>pattern matching</td>
<td>coalgebra</td>
</tr>
<tr>
<td>categorical</td>
<td>algebra</td>
<td></td>
</tr>
</tbody>
</table>