Programming and Reasoning about Coinduction using Copatterns

David Thibodeau
(joint work with A. Abel, B. Pientka, and A. Setzer)

McGill University

March 5, 2013

Infinity in Computer Science

Computer science is bounded by physical limitations: Computations must be finite in time and space.

Yet, some important objects in computer science are, or are modeled as infinite structures

- Functions
- Streams
- I/O devices
- Constantly running processes (e.g. Operating systems)
- etc.

Representing infinity: Existing Solutions

ML: Uses dummy function abstractions to delay, forcing with dummy applications. Hard to work and reason with.

Haskell: Everything is done lazily. There is no difference between finite and infinite. Cannot reason eagerly.

Coq: Coinductive types are non-wellfounded data types. The reduction of cofixpoints is done only under match. Leads to loss of subject reduction. [Gimenez, 1996; Oury, 2008]

Inductive structures

On the other hand, we understand inductive structures very well.

Inductive datatypes are introduced via constructors and eliminated via pattern matching.

```
datatype List : ctype =
| Nil : List
| Cons : Nat → List → List
;

rec append : List → List → List =
fn xs ⇒ fn ys ⇒ case xs of
| Nil ⇒ ys
| Cons z zs ⇒ Cons z (append zs ys)
;
```

New Paradigm: Understand Coinduction as Dual to Induction

We don't understand coinduction through **constructors**, but through **observations**.

Infinite data are impossible to analyse as a whole, hence we can only observe finite parts.

Observations for Functions

Functions are black boxes. We do observations by application of arguments.

$$\frac{M : T \to S \quad N : T}{M\ N : S}$$

We use pattern matching to split the possibly infinite number of different inputs into a finite number of categories.

```
rec isZero : Nat → Bool =
fn e ⇒ case e of
| Zero ⇒ True
| Suc x ⇒ False
;
```

Even if there are infinitely many natural numbers, we only care about if the input is zero or the successor of some natural number.

Observations for Streams

Coinductive objects have defined observations that are provided by the datatype. We define a cofixpoints using **copatterns**.

```
codatatype Stream : ctype =
| Head  : Stream → Nat
| Tail  : Stream → Stream
;
```

Then, we obtain the result of the matching under a **projection copattern**.

```
zeros : Stream
Head zeros = Zero
Tail zeros = zeros
```

Mixing patterns and copatterns

We can redefine the usual constructor.

```
cons : Nat → Stream → Stream
Head (( cons x) y) = x
Tail (( cons x) y) = y
```

We can still use patterns like `(_ x)` and `(_ y)` as **application copatterns**.

The left-hand side is a **composite** copattern. We allow arbitrary mixing of patterns and copatterns.

(Co)Patterns Definition

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

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

Definitions

$$
\begin{array}{rcl}
q_1[f/\cdot] & = & t_1 \\
 & \vdots & \\
q_n[f/\cdot] & = & t_n
\end{array}
$$

Example: Fibonacci Stream

The Fibonacci stream can be define with constructors in the following way.

```
fib = cons 0 (cons 1 (zipWith _+_ fib (tail fib)))
```

It obeys the following recurrence

fib	0	1	1	2	3	5	8
Tail fib	1	1	2	3	5	8	13
zipWith _ + _ fib (Tail fib)	1	2	3	5	8	13	21

Which can be expressed quite nicely with copatterns

```
Head fib = 0
Head ( Tail fib) = 1
Tail ( Tail fib) = zipWith _+_ fib ( Tail fib)
```

This expression satisfies strong normalisation in a system using eager rewriting when matching.

Interactive Program Development

Let us take a function

```
cycleNats : Nat → Stream
```

such that

$$\text{cycleNats } n = n, n-1, \ldots, 1, 0, N, N-1, \ldots, 1, 0, \ldots.$$

e.g.

$$\text{cycleNats } 5 = 5, 4, 3, 2, 1, 0, 2, 1, 0, 2, 1, 0, 2, \ldots$$

How do we construct such function? We can do it interactively

```
cycleNats : Nat → Stream
cycleNats = ?
```

On which we can split on the result (function).

```
cycleNats x = ?
```

Interactive Program Development

We split again on the result (stream).

```
Head ( cycleNats x ) = ?
Tail ( cycleNats x ) = ?
```

Then, we fill the first clause

```
Head ( cycleNats x ) = x
Tail ( cycleNats x ) = ?
```

We do a splitting on x in the second clause.

```
Head ( cycleNats x ) = x
Tail ( cycleNats Zero ) = ?
Tail ( cycleNats (Suc x') ) = ?
```

And we can fill the remaining clauses.

```
Head ( cycleNats x ) = x
Tail ( cycleNats Zero ) = cycleNats N
Tail ( cycleNats (suc x') ) = cycleNats x'
```

Coverage and Progress

This Agda-like interactive program development gives us a notion of coverage.

- Start with the trivial covering. (the copattern · "hole")
- Repeat:
  - Split result or
  - Split a pattern variable
- until you obtain the user-given patterns.

Using such algorithm to obtain covering functions, we can then prove **progress**.

Mixing inductive and coinductive datatypes: Colists

We can also define mutually recursive inductive and coinductive datatypes.

```
codatatype Colist : ctype =
| Out : Colist → Colist'

and datatype Colist' : ctype =
| Nil : Colist'
| Cons : Char → Colist → Colist'
;
```

Say we want to extract all the numbers before the first zero in a stream.

```
mutual
  firstLine : Stream → Colist
  Out (firstLine xs) = firstline' ( Head xs) ( Tail xs)

  firstline' : Char → Stream → Colist'
  firstline' '\n' xs = Nil
  firstline' a xs = Cons a (firstline xs)
```

Contribution

We built a calculus for simple types mixing induction and coinduction in a symmetric way. We achieved the following.

- Subject Reduction
- Coverage Algorithm
- Progress

The next directions for this work leads us to different places

- Strong Normalisation (proof in progress by A. Abel and B. Pientka)
- Extension to Beluga and other dependently types settings

Extending Copatterns to Beluga

Beluga is a two level-system with

- types with dependency from the LF level only.
- function definition and pattern matching using cases and let bindings.

Allows for a good case study for both a dependently typed extension and a foundation for compilation.

Beluga as a foundation syntax for compilation

How do we represent copattern matching outside of this equational style?

```
rec cycleNats : Nat → Stream =
fn x ⇒ cofun Head ⇒ x
           | Tail ⇒ (case x of
                        | Zero ⇒ cycleNats n
                        | Suc x' ⇒ cycleNats x')
```

In the equational style, cycleNats looked like

```
Head (cycleNats x ) = x
Tail (cycleNats Zero ) = cycleNats n
Tail (cycleNats (suc x') ) = cycleNats x'
```

Beluga as a foundation syntax for compilation

If go back to the interactive program development, we realize our Beluga syntax follow it closely

```
rec  cycleNats : Nat → Stream =
fn  x ⇒ cofun Head ⇒ x
            | Tail ⇒ (case  x of
                          | Zero ⇒ cycleNats n
                          | Suc x' ⇒ cycleNats x')
```

We first split on the result. (function)

```
cycleNats x = ?
```

Beluga as a foundation syntax for compilation

If go back to the interactive program development, we realize our Beluga syntax follow it closely

```
rec cycleNats : Nat → Stream =
fn x ⇒ cofun Head ⇒ x
             | Tail ⇒ (case x of
                          | Zero ⇒ cycleNats n
                          | Suc x' ⇒ cycleNats x')
```

Then we split again on the result. (stream)

```
Head ( cycleNats x ) = ?
Tail ( cycleNats x ) = ?
```

Beluga as a foundation syntax for compilation

If go back to the interactive program development, we realize our Beluga syntax follow it closely

```
rec cycleNats : Nat → Stream =
fn x ⇒ cofun Head ⇒ x
            | Tail ⇒ (case x of
                         | Zero ⇒ cycleNats n
                         | Suc x' ⇒ cycleNats x')
```

We conclude by splitting on the pattern variable.

```
Head ( cycleNats x ) = x
Tail ( cycleNats Zero ) = ?
Tail ( cycleNats (Suc x') ) = ?
```

Uses of Cases for Datatypes and Codatatypes

In the equational style, mixing both datatypes and codatatypes requires mutually recursive functions to unpack the observations.

```
mutual
  firstLine : Stream → Colist
  Out (firstLine xs) = firstline’ (Head xs) (Tail xs)

  firstline’ : Char → Stream → Colist’
  firstline’ ‘\n’ xs = Nil
  firstline’ a xs = Cons a (firstline xs)
```

With cases, we can obtain a “simpler” function.

```
rec firstLine : Stream → Colist =
fn e ⇒ cofun Out ⇒ (case Head e of
      | ‘\n’ ⇒ Nil
      | a ⇒ Cons a (firstLine Tail e))
;
```

Dependently Typed Codatatypes

Divergence of lambda terms is proved coinductively.

$$\frac{M \Uparrow}{\lambda x.M \Uparrow} \quad \frac{M \Uparrow}{M\ N \Uparrow} \quad \frac{M \Downarrow \lambda x.M' \quad [N/x]M' \Uparrow}{M\ N \Uparrow}$$

where $M \Uparrow$ means that M diverges,
and $M \Downarrow M'$ means that M evaluates to M'.

```
codatatype Div : [. term] → ctype =
| Div_lam : Div [. lam M] → Div [. M]
| Div_app : Div [. app M N] →
            (Div [. M] + Eval_Div [. M] [. N])

and datatype Eval_Div : [. term] → [. term] → ctype =
| ED : [. eval M M’]→ Div [. M’ N] → Eval_Div [. M] [. N]
;
```

Dependently Typed Datatypes vs Codatatypes

Inductive datatypes refine types

```
datatype : List : [. nat] → ctype =
| Nil : List [. zero]
| Cons : [. nat] → List [. N] → List [. suc N]
;
```

On the other hand, observations are restricted to when codatatypes satisfy a particular type signature.

```
codatatype Div : [. term] → ctype =
| Div_lam : Div [. lam M] → Div [. M]
| Div_app : Div [. app M N] →
            (Div [. M] + Eval_Div [. M] [. N])
```

So if `D : Div [. lam M]`, then `Div_app D` is considered ill-typed.
No such observation can be made!

Impossible Observations

What if a codatatype does not have any possible observation for a given type annotation?

Restrictions on observations from our type dependency imply an idea of coverage.

$$\frac{}{\lambda x.M \Downarrow \lambda x.M} \quad \frac{M \Uparrow}{M\ N \Uparrow} \quad \frac{M \Downarrow \lambda x.M' \quad [N/x]M' \Uparrow}{M\ N \Uparrow}$$

There are no diverging term with a lambda as its head!

```
codatatype Div : [. term] → ctype =
| Div_lam : Div [. lam M] → ?
| Div_app : Div [. app M N] →
            (Div [. M] + Eval_Div [. M] [. N])
```

Impossible Observations

What if a codatatype does not have any possible observation for a given type annotation?

Restrictions on observations from our type dependency imply an idea of coverage.

$$\frac{}{\lambda x.M \Downarrow \lambda x.M} \quad \frac{M \Uparrow}{M\ N \Uparrow} \quad \frac{M \Downarrow \lambda x.M' \quad [N/x]M' \Uparrow}{M\ N \Uparrow}$$

There are no diverging term with a lambda as its head!

```
codatatype Div : [. term] → ctype =
| Div_lam : Div [. lam M] →  Empty
| Div_app : Div [. app M N] →
            (Div [. M] + Eval_Div [. M] [. N])

and datatype Empty → ctype =
;
```

Dependently Typed Copatterns

Theorem

$(\lambda x.x\ x)\ (\lambda x.x\ x)$ *diverges.*

Proof.

The proof is done by coinduction.

We need to reach $(\lambda x.x\ x)\ (\lambda x.x\ x) \Uparrow$ as a **strictly smaller subderivation**.

By our second divergence for application rule, it suffices to show that

- The first term of the application to evaluate to a lambda, $\lambda x.x\ x \Downarrow \lambda x.x\ x$
- The substitution $[(\lambda x.x\ x)/x](x\ x)$ diverges.
  - $[(\lambda x.x\ x)/x](x\ x) \longrightarrow (\lambda x.x\ x)\ (\lambda x.x\ x)$,
  - By coinduction hypothesis, $(\lambda x.x\ x)\ (\lambda x.x\ x) \Uparrow$.

This concludes the proof. □

Dependently Typed Copatterns

Theorem

$(\lambda x.x\ x)\ (\lambda x.x\ x)$ *diverges.*

As a derivation tree, the proof looks like

$$\dfrac{(\lambda x.xx) \Downarrow (\lambda x.x\ x) \qquad \dfrac{(\lambda x.x\ x)\ \Downarrow (\lambda x.x\ x) \qquad \dfrac{\vdots}{(\lambda x.x\ x)\ (\lambda x.x\ x) \Uparrow}}{(\lambda x.x\ x)\ (\lambda x.x\ x) \Uparrow}}{(\lambda x.x\ x)\ (\lambda x.x\ x) \Uparrow}$$

In Beluga, the proof would look like

```
rec OmegaDiverges :
        Div [. app (lam (λx.x x)) (lam (λx.x x))] =
cofun Div_app [. app M N] ⇒
         InR (ED [. eval_lam] OmegaDiverges)
```

Conclusion

- Symmetric calculus mixing inductive and coinductive datatypes;
- Coinduction is modeled using observations instead of constructors;
- Satisfies type preservation and progress;
- Coverage algorithm;
- Proof of strong normalisation in progress;
- Extension to dependent types through Beluga coming soon!