Indexed Copatterns

Reasoning about Infinite Structures by Observations

David Thibodeau   Brigitte Pientka

McGill University

September 24, 2013

Representing Infinite Data

- Plays an important role when reasoning about Input/Output interactions, interactions between server and client, more generally processes
  - Bisimilarity
  - Fairness properties
- Infinite data = circular data:
  - Representing closures when describing evaluation [Tofte, Milner; 1988]
  - Representing circular proofs [Brotherston, 2005]
- Infinite data = diverging computation
  - Diverging small-step evaluation for lambda terms (e.g. Ω)
  - Diverging big step semantics [Leroy and Grall, 2009] mixing finite and infinite computation

How to represent and reason about infinite derivations?

Existing Solutions

General proof systems - lack support for binders

- Coq: loss of subject reduction in the presence of coinduction [Giménez, 1996; Oury, 2008]
- Agda: limitation on definitional equality of coinductive terms

Proof systems supporting binders through Higher Order Abstract Syntax:

- Type theories:
  - Twelf [Harper et al., 1993]: No support for coinduction
  - Beluga [Pientka and Dunfield, 2010]: This talk is about adding support for coinduction and coinductive certified programming
- Proof theory:
  - Abella [Gacek, 2008]: supports coinduction ; no executable program

Current work

Support representing and reasoning about infinite derivations via indexed coinductive datatypes.

New Paradigm: Coinduction as Dual to Induction

We don't understand coinduction through constructors, but through observations (POPL'13, joint work with Andreas Abel, Brigitte Pientka, and Anton Setzer).

In Beluga:

- Inductive datatype

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

- Coinductive datatype

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

The kind **ctype** introduces (co)inductive type.

Induction and Coinduction

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

```
rec append : List → List → List
fn xs ⇒ fn ys ⇒ case xs of
| Nil ⇒ ys
| Cons x xs' ⇒ Cons x (append xs' ys);
```

Induction and Coinduction

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

```
rec append : List → List → List
fn xs ⇒ fn ys ⇒ case xs of
| Nil ⇒ ys
| Cons x xs' ⇒ Cons x (append xs' ys);
```

Coinductive datatypes are eliminated via observations and introduced via copattern matching.

```
rec fib : Stream
observe
| Head ⇒ 0
| Tail Head ⇒ 1
| Tail Tail ⇒ zipWith plus fib (Tail fib);
```

Induction and Coinduction

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

```
rec append : List → List → List
fn xs ⇒ fn ys ⇒ case xs of
| Nil ⇒ ys
| Cons x xs' ⇒ Cons x (append xs' ys);
```

Coinductive datatypes are eliminated via observations and introduced via copattern matching.

```
rec fib : Stream
observe
| Head ⇒ 0
| Tail Head ⇒ 1
| Tail Tail ⇒ zipWith plus fib (Tail fib);
```

The observations on this "observe" value will make it step.

```
Head fib → 0
Head (Tail fib) → 1
Tail (Tail fib) → zipWith plus fib (Tail fib)
```

Previous Contribution (POPL'13)

- A symmetric calculus mixing induction and coinduction in an equational style.
- A coverage algorithm following the style of Agda's interactive mode.
- Proofs of subject reduction and progress.

Contribution

Two examples using indexed codatatypes

- Indexed streams carrying information about sequences of bits inside the stream. (No binders)
- A type-preserving environment-based interpreter where we represent closures coinductively following [Tofte, Milner; 1988].

Goal

Illustrate the idea and usefulness of indexed codatatypes through our prototype in Beluga.

Beluga

2-level proof environment

- Specification level: Logical Framework LF [Harper et al. 1993]
  - Higher Order Abstract Syntax
  - Binders represented by function space
  - Kind **type** introduces LF type.
- Computational level supports inductive and coinductive definitions, recursion and pattern matching
  - Computational types can be indexed by LF terms
  - Kind **ctype** introduces computational (co)datatypes
  - Explicit handling of contexts and substitutions
  - Contextual object: LF term $\mathtt{E}$ is packaged with its surrounding context psi: $[\psi.\mathtt{E}\ ..]$.
    - Context ψ represents all free variables in $\mathtt{E}$.
    - $..$: Identity substitution representing dependency of ψ on $\mathtt{E}$.
    - Binder: $[\psi.\ \mathtt{lam}\ \lambda\mathtt{x.E..x}]$

Indexed (co)datatypes

Indices in datatypes refine types.

```
datatype List : [. nat] → ctype =
| Nil  : List [. z]
| Cons : Bit → List [. N] →List [. s N]
;
```

Indexed (co)datatypes

Indices in datatypes refine types.

```
datatype List : [. nat] → ctype =
| Nil  : List [. z]
| Cons : Bit → List [. N] →List [. s N]
;
```

Indices in codatatypes restrict the type of terms observations can be applied to.

```
codatatype Stream : [. nat] → ctype =
| GetBit  : Stream [. s N] → Bit
| RemBits : Stream [. s N] → Stream [. N]
| Next    : Stream [. z]   → Stream [. N]
;
```

If `l : Stream [. s N]` then `Next l` is not welltyped.

Build Word from Indexed Stream

Suppose we want to extract from a stream of words (e.g. bytes) from our indexed stream of bits.

```
datatype Byte : ctype =
| Nil  : Byte
| Cons : Bit → Byte → Byte;
```

Given a stream observing N inputs return a word (consisting of the N observations) and the remaining stream

```
rec buildByte : {N: [. nat]} Stream [. N] →
                (Byte * Stream [. z]) =
mlam N ⇒ fn s ⇒ case [. N] of
| [. z] ⇒ (Nil , s)
| [. s N] ⇒
  let (bs, s’) = buildByte [. N] (RemBits s) in
  let b = GetBit s in
  (Cons b bs, s’);
```

Input: 0111001000111101001001110111010 1 ...

Output: (01110010, 0011110100100111011101 01 ...)

From Bit Stream to Byte Stream

Then, we can get a stream of words from the indexed stream.

```
codatatype ByteStream : ctype =
| Byte : ByteStream → Byte
| Tail : ByteStream → ByteStream;
```

Given a stream observing N inputs produce a stream of words

```
rec byteStream : {N : [. nat]} Stream [. N] → ByteStream =
mlam N ⇒ fn s ⇒
let (w, s') = buildByte [. N] s in
observe Byte ⇒ w
      | Tail ⇒ byteStream [. N] (Next s');
```

Input: 01110010001111010010011101110101 ...
Output: [01110010, 00111101, 00100111, 01110101, ...]

Simply Typed Lambda Calculus with Fixpoints

$t ::= c \mid T_1 \rightarrow T_2$	Types
$e ::= x \mid e_1\ e_2 \mid \text{abs}\ x.e \mid \text{fix}\ f(x) = e$	Terms

In Beluga, we represent such language in the Logical Framework LF.

```
datatype tp : type =
| arr  : tp → tp → tp
| c    : tp;

datatype tm : tp → type =
| app : tm (arr A B) → tm A → tm B
| abs : (tm A → tm B) → tm (arr A B)
| fix : (tm (arr A B) → tm A → tm B) → tm (arr A B);
```

Operational Semantics with Closures [Tofte, Milner; 1988]

$\phi \vdash e \Downarrow v$ term e steps to value v in environment ϕ.

$$\phi ::= \cdot \mid \phi, (x, v) \qquad \text{Environments}$$
$$v ::= \langle x.e \;;\; \phi \rangle \qquad \text{Values}$$

e in closure depends on x and variables in ϕ.
ϕ defines a value for all free variables in e.
ϕ in closure cl can have reference to cl. Closures might be circular.

$$\frac{x \in \text{Dom } \phi}{\phi \vdash x \Downarrow \phi(x)} \qquad \frac{}{\phi \vdash \text{abs } x.e \Downarrow \langle x.e \;;\; \phi \rangle}$$

$$\frac{cl_\infty = \langle x.e \;;\; \phi, (f, cl_\infty) \rangle}{\phi \vdash \text{fix } f(x) = e \Downarrow cl_\infty}$$

$$\frac{\phi \vdash e_1 \Downarrow \langle x.e \;;\; \phi' \rangle \quad \phi \vdash e_2 \Downarrow v_2 \quad \phi', (x, v_2) \vdash e \Downarrow v}{\phi \vdash e_1 \; e_2 \Downarrow v}$$

Coinductive Closures

These closures being infinite, we need a coinductive definition of values.

$$\phi ::= \cdot \mid \phi, (x, v) \qquad \text{Environments}$$
$$v ::= \langle x.e \; ; \; \phi \rangle \qquad \text{Values}$$

```
schema ctx = tm A;

datatype Env : {ψ:ctx} ctype =
| Empty    : Env [ ]
| Cons     : Val [.A] → Env [ψ]
             → Env [ψ, x:tm A]

and codatatype Val : [.tp] → ctype =
| Val      : Val [.B] → Val' [.B]

and datatype Val' : [.tp] → ctype =
| Closure : [ψ,x:tm A .tm B] → Env [ψ]
             → Val' [.arr A B];
```

Type Preserving Evaluator

```
rec eval : [ψ. tm A] → Env [ψ] → Val [.A] =
fn e ⇒ fn φ ⇒ case e of
```

- ψ links free variables in *e* to variables in ϕ.
- $\phi \vdash e \Downarrow v$ term *e* steps to value *v* in environment ϕ.
- By case analysis on *e*.

Type Preserving Evaluator

rec `eval : [`ψ`. tm A]` $\rightarrow$ `Env [`ψ`]` $\rightarrow$ `Val [.A] =`
fn `e` $\Rightarrow$ **fn** ϕ $\Rightarrow$ **case** `e` **of**
`| [`ψ`. #p .. ]` $\Rightarrow$ `lookup [`ψ`.#p ..]` ϕ

which corresponds to $\dfrac{x \in \mathrm{Dom}\ \phi}{\phi \vdash x \Downarrow \phi(x)}$

`lookup: [`ψ`. tm A]` $\rightarrow$ `Env [`ψ`]` $\rightarrow$ `Val [.A]`

Type Preserving Evaluator

```
rec eval : [ψ. tm A] → Env [ψ] → Val [.A] =
fn e ⇒ fn φ ⇒ case e of
| [ψ. #p .. ] ⇒ lookup [ψ.#p ..] φ
| [ψ. abs (λx.E .. x)] ⇒
  (observe Val ⇒ Closure φ [ψ, x:tm _ . E .. x])
```

which corresponds to $\overline{\phi \vdash \text{abs } x.e \Downarrow \langle x.e \ ; \ \phi \rangle}$

Type Preserving Evaluator

```
rec eval : [ψ. tm A] → Env [ψ] → Val [.A] =
fn e ⇒ fn φ ⇒ case e of
| [ψ. #p .. ] ⇒ lookup [ψ.#p ..] φ
| [ψ. abs (λx.E .. x)] ⇒
  (observe Val ⇒ Closure φ [ψ, x:tm _ . E .. x])
| [ψ. fix (λf. λx.E .. f x)] ⇒
  unfold [ψ, f:tm _ , x:tm _ . E .. f x] φ
```

which corresponds to $\dfrac{cl_\infty = \langle x.e \;;\; \phi, (f, cl_\infty)\rangle}{\phi \vdash \text{fix } f(x) = e \Downarrow cl_\infty}$

where `unfold` creates a reference to itself

```
rec unfold : [ψ,f:tm (arr A B), x:tm A. tm B]
               → Env [ψ] → Val [. (arr A B)] =
fn cl ⇒ fn φ ⇒
(observe Val ⇒ Closure (Cons (unfold cl φ) φ) cl);
```

Type Preserving Evaluator

```
rec eval : [ψ. tm A] → Env [ψ] → Val [.A] =
fn e ⇒ fn φ ⇒ case e of
| [ψ. #p .. ] ⇒ lookup [ψ.#p ..] φ
| [ψ. abs (λx.E .. x)] ⇒
  (observe Val ⇒ Closure φ [ψ, x:tm _ . E .. x])
| [ψ. fix (λf. λx.E .. f x)] ⇒
  unfold [ψ, f:tm _ , x:tm _ . E .. f x] φ
| [ψ. app (E1 ..) (E2 ..)] ⇒
  let Closure φ' [ψ,x:tm _ . E .. x] =
      Val (eval [ψ. E1 ..] φ) in
  let v2 = eval [ψ. E2 ..] φ in
    eval [ψ,x:tm _ . E .. x] (Cons v2 φ')
;
```

which corresponds to

$$\frac{\phi \vdash e_1 \Downarrow \langle x.e\ ;\ \phi' \rangle \quad \phi \vdash e_2 \Downarrow v_2 \quad \phi', (x, v_2) \vdash e \Downarrow v}{\phi \vdash e_1\ e_2 \Downarrow v}$$

Related work

- Coinductive proofs in Abella
  - Bisimulation proofs in π-calculus
  - Attempt to implement closure based interpreter example (broken)
- Operational Semantics in Agda [Danielsson, 2012]
  - Uses partiality monad,
  - No support for binders
- Pretty big-step semantics [Charguéraud, 2012]
  - Eliminates duplication of premisses when dealing with divergence
  - Uses traces to relate inductive and coinductive interpretation of rules

Conclusion

- A prototype for coinductive Beluga supporting inductive and coinductive reasoning in a symmetric fashion
- Examples:
  - Indexed streams keeping track of how much of the current word is still to be read;
  - A type-preserving environment-based interpreter with closures;
  - Reasoning about divergence of lambda terms;
  - Bisimulation of processes in the pi-calculus.
- Moving the idea of copatterns towards dependent types.

Current work

- Extend the meta theory to indexed coinductive types;
- Solve some type reconstruction issues in the implementation;
- Define a notion of coverage for indexed copatterns;
- Extend notion of productivity of [Abel, Pientka; ICFP'13] to indexed types.