CoCaml: Programming with Coinductive Types

Jean-Baptiste Jeannin* Dexter Kozen* Alexandra Silva†

*Cornell University

†Radboud University of Nijmegen and Centrum Wiskunde & Informatica

FSA colloquium, University of Eindhoven
April 18, 2013

Computing with Coalgebraic Data

- Inductive datatypes and functions on those are well-understood; coinductive datatypes often considered difficult to handle, not many programming languages offer the constructs for them.
- OCaml offers the possibility of defining coinductive datatypes, but the means to define recursive functions on them are limited.
- Often the obvious definitions do not halt or provide the wrong solution.
- Even so, there are often perfectly good solutions (examples forthcoming!)
- We show how to extend the language to allow it!

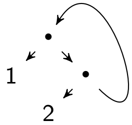
Motivating example

```
type list = N | C of int * list

let rec ones = C(1, ones);; 1,1,1,1,...
let rec alt = C(1, C(2, alt));; 1,2,1,2,...
```

Motivating example

```
type list = N | C of int * list

let rec ones = C(1, ones);; 1,1,1,1,...
let rec alt = C(1, C(2, alt));; 1,2,1,2,...
```

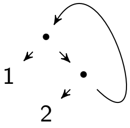
Infinite lists but. . . regular:



Motivating example

```
type list = N | C of int * list

let rec ones = C(1, ones);; 1,1,1,1,...
let rec alt = C(1, C(2, alt));; 1,2,1,2,...
```

Infinite lists but. . . regular:



A simple function:

```
let set l = match l with
| N -> N
| C(h, t) -> (insert h (set t));;
```

We expect `set ones = {1}` and `set alt = {1,2}`.

What is the problem?

- The function definition above will not halt in OCaml...
- even though it is clear what the answer should be;

What is the problem?

- The function definition above will not halt in OCaml...
- even though it is clear what the answer should be;
- Note that this is not a corecursive definition: we are not asking for a greatest solution or a unique solution in a final coalgebra,
- but rather a least solution in a different ordered domain from the one provided by the standard semantics of recursive functions.
- Standard semantics: least solution in the flat Scott domain with bottom element $\perp$ representing nontermination
- Intended semantics: least solution in a different CPO, namely $(\mathcal{P}(\mathbb{Z}), \subseteq)$ with bottom element $\varnothing$.

Motivating example c'd

We would like to use (almost) the same definition and get the intended solution. . .

```
let set l = match l with
| N -> N
| C(h, t) -> (insert h (set t));;
```

Motivating example c'd

We would like to use (almost) the same definition and get the intended solution...

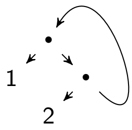
```
let set l = match l with
| N -> N
| C(h, t) -> (insert h (set t));;
```

We change it to:

```
let corec[iterator(N)] set l = match l with
| N -> N
| C(h, t) -> insert h (set t);;
```

The construct `corec` with the parameter `iterator(N)` specifies to the compiler how to solve equations.

For instance, for the infinite list `alt`:



the compiler will generate two equations:

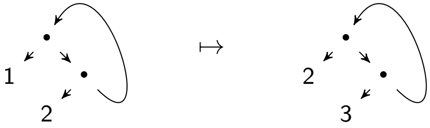
```
set(x) = insert 1 (set(y))
set(y) = insert 2 (set(x))
```

then solve them using `iterator` (least fixed point) which will produce the intended set $\{1, 2\}$.

Motivating example c'd

```
let map f = match arg with
| N -> N
| C(h, t) -> C(f(h), map(f,t));;
```

We would like: `map plusOne alt` to produce the infinite list $2, 3, 2, 3, \ldots$:



This is not a least fixed point computation anymore but rather a solution in the final coalgebra.

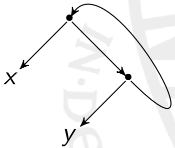
Free variables of a λ-term

```
type term =
  | Var of string            x
  | App of term * term       (f e)
  | Lam of string * term     λx.e

let rec fv = function
  | Var v -> {v}
  | App(t1,t2) -> fv t1 ∪ fv t2
  | Lam(x,t) -> (fv t) - {x}
```

Another Example

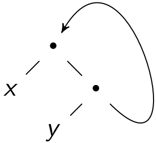
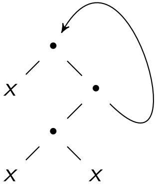
But what about infinitary λ-terms (λ-coterms)?

```
type term =
  | Var of string             x
  | App of term * term        (f e)
  | Lam of string * term      λx.e

let rec fv = function
  | Var v -> {v}
  | App(t1,t2) -> fv t1 ∪ fv t2
  | Lam(x,t) -> (fv t) - {x}

let rec t = App(Var "x", App(Var "y", t))
```

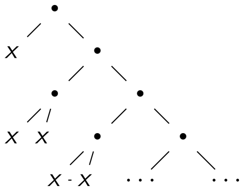


We would like: `fv t = {x,y}` (again LFP).

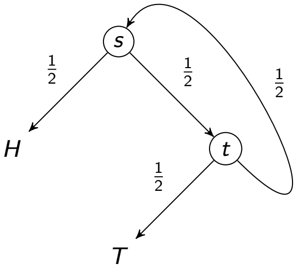
Substitution

Replace y by  in  to get  .

The usual semantics would infinitely unfold the term on the left, generating instead:

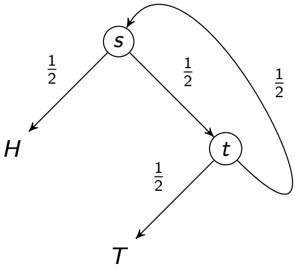


Probabilistic Protocols



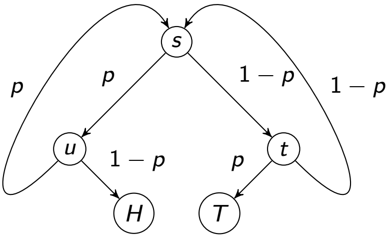
$$\mathrm{Pr}_H(s) = \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} + \cdots = \frac{2}{3}$$
$$\mathrm{Pr}_H(t) = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \cdots = \frac{1}{3}$$

Probabilistic Protocols



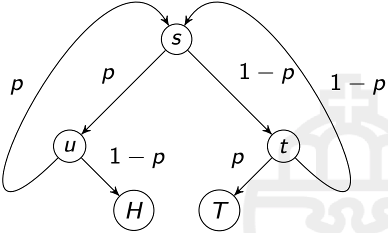
$$\mathrm{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{2} \cdot \mathrm{Pr}_H(t)$$
$$\mathrm{Pr}_H(t) = \tfrac{1}{2} \cdot \mathrm{Pr}_H(s)$$

The Von Neumann Trick



$$\begin{aligned}
\mathrm{Pr}_H(s) &= p \cdot \mathrm{Pr}_H(u) + (1-p) \cdot \mathrm{Pr}_H(t) \\
\mathrm{Pr}_H(u) &= (1-p) + p \cdot \mathrm{Pr}_H(s) \\
\mathrm{Pr}_H(t) &= (1-p) \cdot \mathrm{Pr}_H(s)
\end{aligned}$$

```
type state =
  | H
  | T
  | Flip of float * state * state

let rec pr_heads s = function
  | H -> 1.
  | T -> 0.
  | Flip(p,u,v) ->
    p *. (pr_heads u) +. (1 -. p) *. (pr_heads v)

let rec s = Flip(.345,u,t)
and u = Flip(.345,H,s)
and t = Flip(.345,T,s)

print p_heads s
```



Theoretical Foundations

- Well-founded coalgebras [Taylor 99]
- Recursive coalgebras [Adámek, Lücke, Milius 07]
- Elgot algebras [Adámek, Milius, Velebil 06]
- Corecursive algebras [Capretta, Uustalu, Vene 09]

Ingredients:

- Functor F (usually polynomial or power set)
- domain: an F-coalgebra (C, γ)
- range: an F-algebra (A, α)

$$\begin{tikzcd} C \arrow[r, "h"] \arrow[d, "\gamma"'] & A \\ FC \arrow[r, "Fh"'] & FA \arrow[u, "\alpha"'] \end{tikzcd}$$

Example: Factorial

```
let rec factorial = function
  | 0 -> 1
  | n -> n * factorial (n-1)
```

$$\begin{tikzcd}
\mathbb{N} \arrow[r, "h"] \arrow[d, "\gamma"'] & \mathbb{N} \\
\mathbb{1} + \mathbb{N} \times \mathbb{N} \arrow[r, "\mathrm{id}_{\mathbb{1}} + \mathrm{id}_{\mathbb{N}} \times h"'] & \mathbb{1} + \mathbb{N} \times \mathbb{N} \arrow[u, "\alpha"']
\end{tikzcd}$$

$$FX = \mathbb{1} + \mathbb{N} \times X \qquad \begin{aligned} \gamma(0) &= \iota_0() \\ \gamma(n+1) &= \iota_1(n+1, n) \end{aligned} \qquad \begin{aligned} \alpha(\iota_0()) &= 1 \\ \alpha(\iota_1(n, m)) &= nm \end{aligned}$$

Example: Fibonacci

```
let rec fibonacci = function
  | 0 -> 0
  | 1 -> 1
  | n -> fibonacci (n-1) + fibonacci (n-2)
```

$$\begin{tikzcd}
\mathbb{N} \arrow[rr, "h"] \arrow[d, "\gamma"'] & & \mathbb{N} \\
\mathbb{1} + \mathbb{1} + \mathbb{N} \times \mathbb{N} \arrow[rr, "\mathsf{id}_{\mathbb{1}} + \mathsf{id}_{\mathbb{1}} + h \times h"'] & & \mathbb{1} + \mathbb{1} + \mathbb{N} \times \mathbb{N} \arrow[u, "\alpha"']
\end{tikzcd}$$

$$FX = \mathbb{1} + \mathbb{1} + X \times X \qquad \begin{aligned} \gamma(0) &= \iota_0() \\ \gamma(1) &= \iota_1() \\ \gamma(n+2) &= \iota_2(n+1, n) \end{aligned} \qquad \begin{aligned} \alpha(\iota_0()) &= 0 \\ \alpha(\iota_1()) &= 1 \\ \alpha(\iota_2(n, m)) &= n + m \end{aligned}$$

Example: Quicksort

[Adámek et al 07]

```
let rec partition pivot = function
  | [] -> [], []
  | hd :: tl ->
      let leq, gt = partition pivot tl in
      if hd <= pivot then hd :: leq, gt
      else leq, hd :: gt

let rec quicksort = function
  | [] -> []
  | pivot :: tl ->
      let leq, gt = partition pivot tl in
      (quicksort leq) @ (pivot :: (quicksort gt))
```

$$\begin{tikzcd} A^* \arrow[rr, "h"] \arrow[d, "\gamma"'] & & A^* \\ \mathbb{1} + A^* \times A \times A^* \arrow[rr, "\mathrm{id}_{\mathbb{1}} + h \times \mathrm{id}_A \times h"'] & & \mathbb{1} + A^* \times A \times A^* \arrow[u, "\alpha"'] \end{tikzcd}$$

$$FX = \mathbb{1} + X \times A \times X$$

$$\gamma([\,]) = \iota_0()$$
$$\gamma(\texttt{pivot} :: \texttt{tl}) = \iota_1(\texttt{tl}_{\leq\texttt{pivot}}, \texttt{pivot}, \texttt{tl}_{>\texttt{pivot}})$$

$$\alpha(\iota_0()) = [\,]$$
$$\alpha(\iota_1(\texttt{stl}_{\leq\texttt{pivot}}, \texttt{pivot}, \texttt{stl}_{>\texttt{pivot}})) = \texttt{stl}_{\leq\texttt{pivot}}\ \texttt{@}\ (\texttt{pivot} :: \texttt{stl}_{>\texttt{pivot}})$$

What about Non-Well-Founded Coalgebras?

The foundations existing so far were for unique solutions; we want alternative solutions.

What about Non-Well-Founded Coalgebras?

The foundations existing so far were for unique solutions; we want alternative solutions.

$$\begin{tikzcd} C \arrow[r, "h"] \arrow[d, "\gamma"'] & A \\ FC \arrow[r, "Fh"'] & FA \arrow[u, "\alpha"'] \end{tikzcd}$$

- Even if (C, γ) is not well-founded, the diagram may still have a canonical solution, provided (A, α) comes equipped with a method for solving systems of equations
- The diagram specifies the system to be solved
- The variables are the elements of C and h is their interpretation in A
- The system is finite if C is

The general idea

The programmer specifies the equations as usual with an extra parameter, like in:

```
let corec[iterator(N)] set l = match l with
| N -> N
| C(h, t) -> insert h (set t);;
```

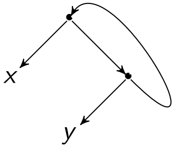
The general idea

The programmer specifies the equations as usual with an extra parameter, like in:

```
let corec[iterator(N)] set l = match l with
| N -> N
| C(h, t) -> insert h (set t);;
```

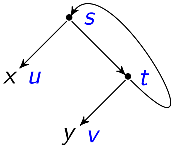
The compiler generates equations and solves them using the extra parameter.

Free Variables of a λ-Coterm

The free variables of  are $\{x, y\}$

Free Variables of a λ-Coterm

The free variables of



are $\{x, y\}$

$$
\begin{aligned}
fv(s) &= fv(u) \cup fv(t) \\
fv(t) &= fv(v) \cup fv(s) \\
fv(u) &= \{x\} \\
fv(v) &= \{y\}
\end{aligned}
$$

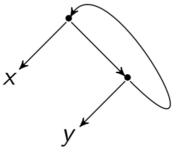
The least solution in $(\mathcal{P}(\mathrm{Var}), \subseteq)$ is $\{x, y\}$

Standard semantics: $A \cup \perp = \perp$, whereas here $A \cup \varnothing = A$

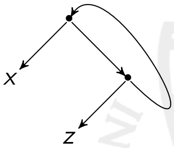
Substitution

```
let corec[constructor] subst x t = match arg with
 | Var v
-> if (v = x) then t else Var v
 | App(t1, t2)
-> App(subst (x, t, t1), subst (x, t, t2));;
```

Replace y by z in



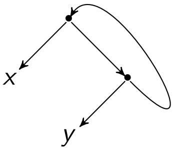
to get



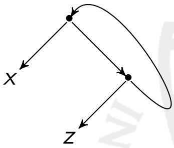
Substitution

```
let corec[constructor] subst x t = match arg with
 | Var v
-> if (v = x) then t else Var v
 | App(t1, t2)
-> App(subst (x, t, t1), subst (x, t, t2));;
```

Replace y by z in



to get

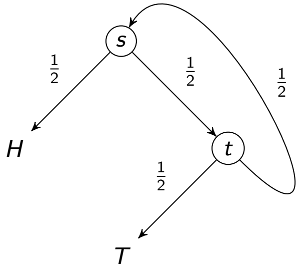


We would again get 4 equations in 4 unknowns

In this case the solution is unique—the algebra is the final coalgebra

Standard semantics: not the unique solution in the final coalgebra C, but the least solution in a Scott domain $C_\perp$

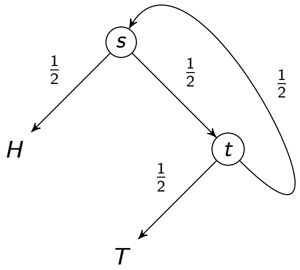
Example: Probabilistic Protocols



$$\mathrm{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{2} \cdot \mathrm{Pr}_H(t) \qquad \mathrm{Pr}_H(t) = \tfrac{1}{2} \cdot \mathrm{Pr}_H(s)$$

- Can calculate expected running times, higher moments, outcome functions similarly
- These are all least solutions in an appropriate ordered domain—in the above example, $([0,1], \leq)$

Probabilistic Protocols



$$\mathsf{E}(s) = \tfrac{1}{2} \cdot 1 + \tfrac{1}{2} \cdot (1 + \mathsf{E}(t)) = 1 + \tfrac{1}{2} E(t)$$
$$\mathsf{E}(t) = \tfrac{1}{2} \cdot 1 + \tfrac{1}{2} \cdot (1 + \mathsf{E}(s)) = 1 + \tfrac{1}{2} E(s)$$

- Least solution in $\mathbb{R}_+ \cup \{\infty\}$ is $\mathsf{E}(s) = \mathsf{E}(t) = 2$
- Also the unique bounded solution, because the fixpoint equation is contractive

Other Non-Well-Founded Examples

- static analysis, abstract interpretation
- p-adic arithmetic
- automata constructions

Implementation

- We implemented `corec` constructor which takes a solver as a parameter
- We implemented several general solvers: least fixed point, unique solution in a final coalgebra, gaussian elimination, ...

Implementation

- We implemented `corec` constructor which takes a solver as a parameter
- We implemented several general solvers: least fixed point, unique solution in a final coalgebra, gaussian elimination, ...
- Solvers are implemented directly in the interpreter, as transformers from an abstract syntax tree to another abstract syntax tree.
- Future: to provide tools to manipulate the abstract syntax tree allowing programmers to easily specify their solver.

Conclusions

- CoCaml offers new program constructs and functionalities to implement functions on coinductive structures.
- Examples illustrate the need for new constructs
- New constructs enable allow definitions very much in the style of standard recursive functions.

http://www.cs.cornell.edu/Projects/CoCaml/

Thanks!