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
• 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

```ocaml
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,...
```

A simple function:

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

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

```ocaml
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:

1

2
```

A simple function:

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

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

Alexandra Silva
April 18 2013
CoCaml: Programming with Coinductive Types
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 bottom representing nontermination
- Intended semantics: least solution in a different CPO, namely $(\mathcal{P}(\mathbb{Z}), \subseteq)$ with bottom element $\emptyset$. 
We would like to use (almost) the same definition and get the intended solution...

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

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

We change it to:

```ocaml
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 $\textit{alt}$:

\[
\begin{align*}
\text{set}(x) & = \text{insert } 1 \ (\text{set}(y)) \\
\text{set}(y) & = \text{insert } 2 \ (\text{set}(x))
\end{align*}
\]

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

\[
\begin{array}{c}
1 \\
\downarrow \\
2
\end{array}
\]
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, ...:

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

Free variables of a $\lambda$-term

type term =
  | Var of string $x$
  | App of term * term $(f \ e)$
  | Lam of string * term $\lambda x.e$

let rec fv = function
  | Var v -> \{v\}
  | App(t1,t2) -> fv t1 \cup fv t2
  | Lam(x,t) -> (fv t) - \{x\}
But what about infinitary $\lambda$-terms ($\lambda$-coterms)?

type term =
    | Var of string         $x$
    | App of term * term    ${(f \ e)}$
    | Lam of string * term  $\lambda x.e$

let rec fv = function
    | Var v -> {v}
    | App(t1,t2) -> fv t1 $\cup$ 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 $\bullet$ in $x \bullet x$ to get $x \bullet x$.

The usual semantics would infinitely unfold the term on the left, generating instead:
\[
\begin{align*}
\Pr_H(s) &= \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} + \cdots = \frac{2}{3}, \\
\Pr_H(t) &= \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \cdots = \frac{1}{3}
\end{align*}
\]
\[Pr_H(s) = \frac{1}{2} + \frac{1}{2} \cdot Pr_H(t)\]
\[Pr_H(t) = \frac{1}{2} \cdot Pr_H(s)\]
The Von Neumann Trick

\[
Pr_H(s) = p \cdot Pr_H(u) + (1 - p) \cdot Pr_H(t)
\]

\[
Pr_H(u) = (1 - p) + p \cdot Pr_H(s)
\]

\[
Pr_H(t) = (1 - p) \cdot Pr_H(s)
\]
The Von Neumann Trick

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 $(C, \gamma)$
- range: an $F$-algebra $(A, \alpha)$

\[
\begin{array}{c}
C \xrightarrow{\gamma} FC \\
\downarrow{h} \quad \downarrow{Fh} \\
A \quad FA \\
\uparrow{\alpha} \quad \uparrow{\gamma}
\end{array}
\]
Example: Factorial

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

\[ FX = 1 + \mathbb{N} \times X \]
\[ \gamma(0) = \nu_0() \]
\[ \gamma(n + 1) = \nu_1(n + 1, n) \]
\[ \alpha(\nu_0()) = 1 \]
\[ \alpha(\nu_1(n, m)) = nm \]
Example: Fibonacci

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

\[
\begin{align*}
\gamma &: \mathbb{N} \to \mathbb{N} \\
\alpha &: \mathbb{N} \to \mathbb{N} \\
F \times X &= \mathbb{N} \\
\end{align*}
\]

\[
\begin{align*}
\gamma(0) &= \nu_0() \\
\gamma(1) &= \nu_1() \\
\gamma(n + 2) &= \nu_2(n + 1, n) \\
\end{align*}
\]

\[
\begin{align*}
\alpha(\nu_0()) &= 0 \\
\alpha(\nu_1()) &= 1 \\
\alpha(\nu_2(n, m)) &= n + m \\
\end{align*}
\]
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))
Example: Quicksort

[Adámek et al 07]

\[ A^* \xrightarrow{h} A^* \]
\[ \gamma \downarrow \quad \downarrow \gamma \]
\[ 1 + A^* \times A \times A^* \xrightarrow{\text{id}_1 + h \times \text{id}_A \times h} 1 + A^* \times A \times A^* \]

\[ FX = 1 + X \times A \times X \]

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

\[ \alpha(\iota_0()) = [ ] \]
\[ \alpha(\iota_1(\text{stl} \leq \text{pivot}, \text{pivot}, \text{stl} > \text{pivot})) = \text{stl} \leq \text{pivot} \circ (\text{pivot :: stl} > \text{pivot}) \]
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.

Even if \((C, \gamma)\) is not well-founded, the diagram may still have a canonical solution, provided \((A, \alpha)\) 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:

```ocaml
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:

```ocaml
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 $\lambda$-Coterm

The free variables of

$\text{fv}(s) = \text{fv}(u) \cup \text{fv}(t)$

$\text{fv}(t) = \text{fv}(v) \cup \text{fv}(s)$

$\text{fv}(u) = \{x\}$

$\text{fv}(v) = \{y\}$

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

Standard semantics: $A \cup \bot = \bot$, whereas here $A \cup \emptyset = A$
The free variables of $s$ are \{x, y\}

\[ \text{fv}(s) = \text{fv}(u) \cup \text{fv}(t) \]
\[ \text{fv}(t) = \text{fv}(v) \cup \text{fv}(s) \]
\[ \text{fv}(u) = \{x\} \]
\[ \text{fv}(v) = \{y\} \]

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

Standard semantics: $A \cup \bot = \bot$, whereas here $A \cup \emptyset = A$
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

\[
\begin{array}{c}
\text{to get}
\end{array}
\]

\[
\begin{array}{c}
\text{x} \\
\downarrow
\end{array}
\]

\[
\begin{array}{c}
\text{y} \\
\downarrow
\end{array}
\]

\[
\begin{array}{c}
\text{x} \\
\downarrow
\end{array}
\]

\[
\begin{array}{c}
\text{z} \\
\downarrow
\end{array}
\]
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

\[ x \rightarrow y \rightarrow \bullet \rightarrow x \]

to get

\[ x \rightarrow \bullet \rightarrow z \rightarrow x \]

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 \textit{unique} solution in the final coalgebra $C$, but the \textit{least} solution in a Scott domain $C_\bot$. 
Example: Probabilistic Protocols

Pr_H(s) = \frac{1}{2} + \frac{1}{2} \cdot Pr_H(t) \quad Pr_H(t) = \frac{1}{2} \cdot 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)
\[ E(s) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 + E(t)) = 1 + \frac{1}{2} E(t) \]
\[ E(t) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 + E(s)) = 1 + \frac{1}{2} E(s) \]

- Least solution in \( \mathbb{R}_+ \cup \{\infty\} \) is \( E(s) = E(t) = 2 \)
- 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
We implemented \texttt{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, ...
• 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!