Rational Fixpoints in Programming Languages

Alexandra Silva†

†Radboud University of Nijmegen and Centrum Wiskunde & Informatica

Fixed points in Computer Science (FICS)
September 1, 2013
Research is not a lonely business

Jean-Baptiste Jeannin  dreaming high  Dexter Kozen
Research is not a lonely business

Jean-Baptiste Jeannin  dreaming high  Dexter Kozen

Stefan Milius  Larry Moss
• Inductive datatypes vs coinductive datatypes.
• 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!
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,...
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
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 $\bot$ representing nontermination
- Intended semantics: least solution in a different CPO, namely $\langle \mathcal{P}(\mathbb{Z}), \subseteq \rangle$ with bottom element $\emptyset$. 
We would like to use (almost) the same definition and get the intended solution...

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

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 \( alt \):

\[
\begin{aligned}
1 & \rightarrow 2 \\
2 & \rightarrow 1 \\
\end{aligned}
\]

the compiler will generate two equations:

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

then solve them using iterator (least fixed point) which will produce the intended set \( \{1, 2\} \).
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.
• Regular/rational coinductive objects are not graphs!
A Caveat

- Regular/rational coinductive objects are not graphs!
- they are rational elements of a final coalgebra
- rational = regular = has a finite representation
- functions defined on them must be independent of the representation
Free variables of a \( \lambda \)-term

```ml
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\}
```

Another Example

Excerpt from: Rational Fixpoints in Programming Languages

Alexandra Silva

FICS 2013
But what about infinitary $\lambda$-terms ($\lambda$-coterm)?

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 \( x \) in \( \cdot \cdot \cdot \) to get \( \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \).

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

\[ \Pr_H(s) = \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128} + \cdots = \frac{2}{3} \]

\[ \Pr_H(t) = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \cdots = \frac{1}{3} \]
Probabilistic Protocols

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

\[
\Pr_H(t) = \frac{1}{2} \cdot \Pr_H(s)
\]
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 $(A, \alpha)$
Example: Factorial

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

\[
\begin{align*}
\mathbb{N} &\xrightarrow{h} \mathbb{N} \\
1 + \mathbb{N} \times \mathbb{N} &\xrightarrow{id_1 + id_\mathbb{N} \times h} 1 + \mathbb{N} \times \mathbb{N}
\end{align*}
\]

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

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

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

\[
\begin{align*}
A^* & \xrightarrow{h} A^* \\
\gamma & \downarrow \\
\mathbb{1} + A^* \times A \times A^* & \xrightarrow{id_{\mathbb{1}} + h \times \text{id}_A \times h} \mathbb{1} + A^* \times A \times A^*
\end{align*}
\]

\[
FX = \mathbb{1} + X \times A \times X
\]

\[
\begin{align*}
\gamma([\,]) &= \iota_0() \\
\gamma(\text{pivot} :: \text{tl}) &= \iota_1(\text{tl}_{\leq \text{pivot}}, \text{pivot}, \text{tl}_{> \text{pivot}}) \\
\alpha(\iota_0()) &= [\,] \\
\alpha(\iota_1(\text{stl}_{\leq \text{pivot}}, \text{pivot}, \text{stl}_{> \text{pivot}})) &= \text{stl}_{\leq \text{pivot}} @ (\text{pivot} :: \text{stl}_{> \text{pivot}})
\end{align*}
\]
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.

```
C --h--> A
|   ↑     |
|  γ    |  α  |
|   ↓   |     |
FC --> FA
```

- 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:

```haskell
let corec[iterator(N)] set l = match l with
| N  -> N
| C(h, t) -> insert h (set t);;
```
The programmer specifies the equations as usual with an extra parameter, like in:

```plaintext
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 $s$ are $\\{x, y\}$.

$$fv(s) = fv(u) \cup fv(t)$$

$$fv(t) = fv(v) \cup fv(s)$$

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

$$fv(v) = \{y\}$$
Free Variables of a $\lambda$-Coterm

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

The free variables of $u$ are $\{x\}$.

The free variables of $t$ are $\{y\}$.

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

\[
\text{let corec[constructor] subst x t = match arg with}
\]
\[
| \text{Var v} \\
| \text{if (v = x) then t else Var v} \\
| \text{App(t1, t2)} \\
| \text{App(subst (x, t, t1), subst (x, t, t2))};
\]

Replace $y$ by $z$ in

\[
\begin{align*}
x & \quad \downarrow \\
y & \quad \downarrow \\
\end{align*}
\]

to get

\[
\begin{align*}
x & \quad \downarrow \\
z & \quad \downarrow \\
\end{align*}
\]
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

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

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

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

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

- 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
CoCaml Implementation

- CoCaml interpreter with `let corec [solver]` – allows the programmer to specify a solver
- CoCaml has three built-in general-purpose solvers:
  - `iterator(init)` (iterate to fixpoint)
  - `constructor` (build unique solution in a final coalgebra)
  - `gaussian` (solve linear equations)
- Interface for the programmer to build custom solvers
module type Solver = sig
  type var
  type expr
  type t
  val fresh : unit -> var
  val unk : var -> expr
  val solve : var -> (var * expr) list -> t
end

• \texttt{var} = type of the variables in the equations, also the type of the left-hand sides of equations

• \texttt{expr} = type of the right-hand sides of the equations

• \texttt{t} = return type of the solver, also of the function that is being defined
Beyond regular/rational fixpoints

- \((1, 2, 3, 4, \ldots)\)
- Is this a regular/rational object?
• (1, 2, 3, 4, …)
• Is this a regular/rational object?

yes... but in a different category: Vect.

We are now extending our work to cover rational fixpoints in Vect and other categories.
Rational/regular fixpoints in other languages

- In Haskell: Ghani/Uustalu, Oliveira, Trancon y Widemann, ... 
- In Prolog and Java: Ancona and Zucca.
Conclusions

- CoCaml offers **new programming constructs and functionality** to implement recursive functions on infinite regular coinductive structures.
- We have **lots of examples** to illustrate the usefulness of the new constructs.
- One can define recursive functions on regular infinite coinductive data in a call-by-value language in the same style and with the same elegance as recursive functions on inductive data.
Thanks!

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