Codata types and Copattern matching

Yann Régis-Gianas, Paul Laforgue

Paris Diderot University, France

August 08, 2016

Motivations

- Finite
  - Structures : List, Tree...
  - Inductive types and pattern matching.
- Infinite
  - Structures : Stream, Infinite tree...
  - ???

Motivations

- Finite
  - Structures : List, Tree...
  - Inductive types and pattern matching.
- Infinite
  - Structures : Stream, Infinite tree...
  - Coinductive types and copattern matching !

Copatterns : Programming Infinite Structures by Observations.
Abel, Pientka, Thibodeau and Setzer (POPL – 2013)

Data types and Pattern matching

A data type is defined by its Constructors :

$$\textbf{type}\ \text{‘a list} = \text{Nil} \mid \text{Cons}\ \textbf{of}\ \text{‘a} \times \text{‘a list}$$

$$\textbf{let}\ \textit{ns} : \text{int list} = \text{Cons}\,(1, \text{Cons}\,(2, \text{Nil}))$$

Deconstruct with **pattern matching** :

$$\begin{array}{l} \textbf{let rec}\ \text{map}\ f\ \textit{xs} = \textbf{match}\ \textit{xs}\ \textbf{with} \\ \quad \mid \text{Nil} \qquad\qquad\quad \to \text{Nil} \\ \quad \mid \text{Cons}\,(x, \textit{xs}) \to \text{Cons}\,(f\ x, \text{map}\ f\ \textit{xs}) \end{array}$$

$$\text{map succ}\ \textit{ns};;$$

$$- : \text{int list} = \text{Cons}\,(2, \text{Cons}\,(3, \text{Nil}))$$

Motivations

Problems when handling infinite structures in a call-by-value evaluation strategy :

- Call-by-value is an evaluation strategy in which the arguments are evaluated before being passed to the functions.

  let rec zeros = Cons $(0,$ zeros$)$

  $\Rightarrow$ the evaluation of map succ zeros diverges.
- Solution : ?

Motivations

Problems when handling infinite structures in a call-by-value evaluation strategy :

- Call-by-value $\Rightarrow$ divergence.
- Solution : simulate call-by-name (à la Haskell). Call-by-name is a strategy in which the arguments are not evaluated before the function is called. Use **thunks** to differ the evaluation of the tail.

```
type 'a list = Nil | Cons of 'a × (unit → 'a list)
let rec zeros = Cons (0, fun () → zeros)
let rec map f xs = match xs with
    | Nil → Nil
    | Cons (a, th) → Cons (f a, fun () → map f (th ()))

map succ zeros;;
 − : int list = Cons (1, <fun>)
```

Motivations

Problems when handling infinite structures in a call-by-value evaluation strategy :

- Call-by-value is an evaluation strategy in which the arguments are evaluated before being passed to the functions.
- Solution : simulate call-by-name, using thunks.
- Question : alternative solution ?

Codata types and Copattern matching

Codata types are defined by their destructors.

$\textbf{cotype}\ (\text{'a}, \text{'b})\ \text{product} = \{\, \text{fst} : \text{'a}; \text{snd} : \text{'b} \,\}$

Introduction with **copattern matching**.

$\textbf{let}\ \text{pair} : (\text{int}, \text{char})\ \text{product} = \textbf{cofix}\ \text{pair}\ \textbf{with}$
$\quad |\ \text{pair.fst} \rightarrow 1$
$\quad |\ \text{pair.snd} \rightarrow \texttt{'n'}$

Codata types and Copattern matching

$$\textbf{cotype}\ \text{'a stream} = \{\text{head} : \text{'a}; \text{tail} : \text{'a stream}\}$$

$$\begin{array}{l}\textbf{let}\ \text{zeros} : \text{int stream} = \textbf{cofix}\ \text{zeros}\ \textbf{with} \\ \quad \mid \text{zeros.head} \rightarrow 0 \\ \quad \mid \text{zeros.tail} \quad \rightarrow \text{zeros}\end{array}$$

Codata types and Copattern matching

$\textbf{cotype}\ \text{`a stream} = \{\text{head} : \text{`a}; \text{tail} : \text{`a stream}\}$

$\textbf{let}\ \text{from} : \text{int} \to \text{int stream} = \textbf{cofix}\ \text{from}\ \textbf{with}$
$\quad |\ (\text{from}\ n).\text{head} \to n$
$\quad |\ (\text{from}\ n).\text{tail} \quad \to \text{from}\ (\text{succ}\ n)$

$(\text{from}\ 3).\text{head} = 3$
$(\text{from}\ 3).\text{tail} = <\!\textit{cofun}\!>$
$(\text{from}\ 3).\text{tail}.\text{head} = 4$
... and so on

Cergy : a *p.o.c* programming language

We implemented a core programming language, Cergy.

- purely functional and statically typed
- has data types and pattern matching
- has codata types and copattern matching
- has an abstract machine

An untyped semantics

- Abel et al. provided a *typed* semantics : "*Whether an expression is considered a value or not depends also on its type*"
- We give an *untyped* semantics for the same language, in which values do not carry types.

Repeated code is annoying

$$
\begin{array}{l}
\textbf{let rec } \mathsf{map}\ f\ xs = \textbf{match } xs \textbf{ with} \\
\quad \mid \mathsf{Nil} \qquad\qquad\quad \to \mathsf{Nil} \\
\quad \mid \mathsf{Cons}\ (x, xs) \to \mathsf{Cons}\ (\mathsf{f\ x}, \mathsf{map\ f\ xs})
\end{array}
$$

$$
\begin{array}{l}
\textbf{let rec } \mathsf{qmap} : (\text{‘a} \to \text{‘b}) \to \text{‘a } \mathsf{stream} \to \text{‘b } \mathsf{stream} = \\
\quad \textbf{cofix } \mathsf{qmap} \textbf{ with} \\
\quad \mid (\mathsf{qmap}\ f\ s).\mathsf{head} \to \mathsf{f\ s.head} \\
\quad \mid (\mathsf{qmap}\ f\ s).\mathsf{tail} \to \mathsf{qmap\ f\ s.tail}
\end{array}
$$

- Our technical contributions :
  - A proof-of-concept programming language.
  - We provide an untyped small-step semantics and an abstract machine.
  - We compile copatterns to efficient tries (not shown here).
- Future work
  - Memoization and cofunctions.
  - Extending copatterns to OCaml.
    Bring codata types and copattern matching out of the context of proof assistants.
  - Prove our semantics.
  - Extend the scope of application for copatterns.
    How many use cases await to be discovered ?