Productive Coprogramming with Guarded Recursion

Jeroen Slot
Ilse Pool

Type Theory and Coq

May 7, 2019

Overview

Important from previous presentations

- Co-recursive functions
- Guard condition: a definition is only accepted if all recursive calls occur inside a constructor.
- Coq uses guardesness checkers to ensure that the guard condition is satisfied.

Example

```
CoFixpoint repeat (A:Set)(a:A) : LList A :=
  LCons a (repeat a).
```

Productivity

- Even if a program generates an infinite amount of data, each piece will be generated in finite time.
- Productivity is guaranteed by guardedness.
- We use:
  - **data** Stream = StreamCons Integer Stream
  - StreamCons :: Integer → Stream → Stream

Examples

ones

ones :: Stream
ones = StreamCons 1 *ones*

filter

filter :: (Integer → Bool) → Stream → Stream
filter f(StreamCons z s) =
  if f z then StreamCons z (*filter* f s)
         else *filter* f s

Why guardedness checks are not enough

mergef

```
mergef :: (Integer → Integer → Stream → Stream) →
          Stream → Stream → Stream
mergef f(StreamCons x xs)(StreamCons y ys) =
    f x y (mergef f xs ys)
```

- Not guarded: no constructors.
- *badf* :: Integer → Integer → Stream → Stream
  badf x y s = s

Why guardedness checks are not enough

mergef

```
mergef :: (Integer → Integer → Stream → Stream) →
          Stream → Stream → Stream
mergef f(StreamCons x xs)(StreamCons y ys) =
    f x y (mergef f xs ys)
```

- We want to make f return "instructions" on how to transform the stream back to *mergef*, which then executes them in a way that the guardedness checker can see it is guarded.
- f :: Integer → Integer → Integer
- f :: Integer → Integer → (Integer, [Integer])
- However, it is difficult to see if we can capture all possibilities for good functions in a single type.

Solution: a guardedness type constructor

- We introduce a guardedness type constructor that may only be used in a guarded way, denoted by $\triangleright$.
- Applied to a type A, we write it as $\triangleright$A: a value of A is only available "tomorrow".
- Only a constructor can bridge the gap between "today" and "tomorrow".
- f :: Integer $\rightarrow$ Integer $\rightarrow$ $\triangleright$GStream $\rightarrow$ Stream

Guardedness type constructor

- StreamCons :: Integer $\rightarrow \triangleright$GStream $\rightarrow$ Stream
- deStreamCons :: Stream $\rightarrow$ (Integer,$\triangleright$GStream)

fix, **pure** and $\circledast$

- **fix** :: $(\triangleright A \to A) \to A$
  Example: *ones* = fix(StreamCons 1)
- **pure** :: $A \to \triangleright A$
- $\circledast$ :: $\triangleright(A \to B) \to \triangleright A \to \triangleright B$

Productive *mergef*

mergef

$$
\begin{aligned}
&\mathit{mergef} :: (\mathsf{Integer} \to \mathsf{Integer} \to \triangleright\mathsf{GStream} \to \mathsf{Stream}) \to \\
&\qquad \mathsf{Stream} \to \mathsf{Stream} \to \mathsf{Stream} \\
&\mathit{mergef}\ \mathsf{f} = \mathbf{fix}\ (\lambda g\ \mathit{xs}\ \mathit{ys}. \\
&\qquad\qquad \mathbf{let}\ (\mathsf{x},\mathsf{xs}') = \mathsf{deStreamCons\ xs} \\
&\qquad\qquad\quad\ (\mathsf{y},\mathsf{ys}') = \mathsf{deStreamCons\ ys} \\
&\qquad\qquad \mathbf{in}\ \mathsf{f\ x\ y}\ (\mathsf{g} \circledast \mathsf{xs}' \circledast \mathsf{ys}'))
\end{aligned}
$$

From the infinite to the finite

- Cosider the function that reads an finite prefix of an infinite stream into a finite list.

take

```
take :: Natural → Stream → [Integer]
take 0 s = []
take (n+1) s = x : take n s'
   where (x,s') = deStreamCons s
```

- Recall: deStreamCons :: Stream → (Integer,▷GStream)
- s' has type ▷GStream, but must have type Stream.
- Once a stream has been fully constructed, it should not matter when each part was constructed.

Clock variables

- A clock variable κ represents an individual time sequence that can be used for safe construction of infinite data, like streams.
- We write $\triangleright^{\kappa}$A to indicate which time stream we are considering.
- StreamCons$^{\kappa}$ :: Integer $\rightarrow \triangleright^{\kappa}$GStream$^{\kappa} \rightarrow$ GStream$^{\kappa}$
- deStreamCons$^{\kappa}$ :: GStream$^{\kappa} \rightarrow$ (Integer,$\triangleright^{\kappa}$GStream$^{\kappa}$)

Clock variables

- $\mathsf{GStream}^{\kappa}$: the type of infinite streams in the process of construction.
- $\forall\kappa.\mathsf{GStream}^{\kappa}$: a finished infinite stream.

- **fix** :: $\forall\kappa.(\triangleright^{\kappa}\mathsf{A} \to \mathsf{A}) \to \mathsf{A}$
- **pure** :: $\forall\kappa.\mathsf{A} \to \triangleright^{\kappa}\mathsf{A}$
- $\circledast$ :: $\forall\kappa.\triangleright^{\kappa}(\mathsf{A} \to \mathsf{B}) \to \triangleright^{\kappa}\ \mathsf{A} \to \triangleright^{\kappa}\ \mathsf{B}$

- **force** :: $(\forall\kappa.\triangleright^{\kappa}\mathsf{A}) \to (\forall\kappa.\mathsf{A})$

Example **force**

Example

deStreamCons :: $(\forall\kappa.\mathsf{GStream}^{\kappa}) \rightarrow (\mathsf{Integer}, \forall\kappa.\mathsf{GStream}^{\kappa})$
deStreamCons x =
    $(\Lambda\kappa.\mathbf{fst}\ (\mathsf{deStreamCons}^{\kappa}\ (\mathsf{x}[\kappa])),$
      $\mathbf{force}\ (\Lambda\kappa.\mathbf{snd}\ (\mathsf{deStreamCons}^{\kappa}\ (\mathsf{x}[\kappa]))))$

Well-typed *take*

take

$$take :: \text{Natural} \to \forall \kappa.\text{GStream}^{\kappa} \to [\text{Integer}]$$
$$take\ 0\ s = []$$
$$take\ (n+1)\ s = x : take\ n\ s'$$
$$\textbf{where}\ (x,s') = \text{deStreamCons}\ s$$

(Final) Coalgebras

unfold

$unfold :: (A \to \mathbb{N} \times A) \to A \to \forall \kappa . \mathsf{GStream}^{\kappa}$

The use of clock variables

map

map f s = StreamCons (f x) (*map* f s')
  where (x,s') = deStreamCons s

maap

maap f (StreamCons x (StreamCons y s''))
  StreamCons (f x) (StreamCons (f y) (*maap* f s''))

nats

nats = StreamCons 0 (*map* (λx.x+1) *nats*)

naats

naats = StreamCons 0 (*maap* (λx.x+1) *naats*)

Different behaviour of extensionally equal *map* and *maap*

map

map f s = StreamCons (f x) (*map* f s')
  where (x,s') = deStreamCons s

maap

maap f (StreamCons x (StreamCons y s''))
  StreamCons (f x) (StreamCons (f y) (*maap* f s''))

map

$map :: \forall\kappa.(\text{Integer} \to \text{Integer}) \to \text{GStream}^{\kappa} \to \text{GStream}^{\kappa}$

maap

$maap :: (\text{Integer} \to \text{Integer}) \to (\forall\kappa.\text{GStream}^{\kappa}) \to (\forall\kappa.\text{GStream}^{\kappa})$

The end