# Coalgebraic Programming Using Copattern Matching

Anton Setzer

Swansea University, Swansea UK

Gregynog, Wales, UK, 27 June 2013

Continuity, Computability, Constructivity –
From Logic to Algorithms (CCC 2013)

# Treating Real Numbers as Acclimatised Real Numbers

- We want to formulate real numbers in dependent type theory.
- Instead of working with concrete computable real numbers we want to work with
  - axiomatized abstract real numbers,
  - and a predicate for real numbers being computable.
- Then we show that functions we want to define map computable real numbers to computable ones.
- From this we obtain an algorithm for computing the function on suitable representations.

# Postulates

- The theorem prover Agda has the concept of a postulate.
- $\text{postulate } a : A$
  means that we introduce a new constant $a$ of type $A$ without any computation rules.
- As in any axiomatic approach, postulates can make Agda inconsistent:

$$\text{postulate falsum} : \bot$$

  allows us to prove everything.
- Postulates are okay, if one allows them in a restricted way.

# Real Number Axioms in Agda

$$
\begin{array}{llll}
\text{postulate} & \mathbb{R} & : & \text{Set} \\
\text{postulate} & \text{zero} & : & \mathbb{R} \\
\text{postulate} & \_+\_ & : & \mathbb{R} \to \mathbb{R} \to \mathbb{R} \\
\text{postulate} & \text{ax+} & : & (r : \mathbb{R}) \to r + 0 == r \\
\ldots & & &
\end{array}
$$

# Signed Digit Reals

$$
\begin{aligned}
&\mathrm{Digit} = \{-1, 0, 1\} : \mathrm{Set} \\
&\mathrm{codata\ SignedDigit} : \mathbb{R} \to \mathrm{Set\ \ where} \\
&\quad \mathrm{signedDigit} : (r : \mathbb{R}) \\
&\qquad\qquad\qquad \to (r \in [-1, 1]) \\
&\qquad\qquad\qquad \to (d : \mathrm{Digit}) \\
&\qquad\qquad\qquad \to \mathrm{SignedDigit}\ (2 * r - d) \\
&\qquad\qquad\qquad \to \mathrm{SignedDigit}\ r
\end{aligned}
$$

We can extract from a proof of SignedDigit the nth Digit:

$$
\mathrm{signedDigit\_to\_nthDigit} : (r : \mathbb{R}) \to (\mathrm{SignedDigit}\ r) \to \mathbb{N} \to \mathrm{Digit}
$$

# Required Property Needed

- We want that if we prove for some $r$

  $$p : \mathrm{SignedDigit}\ r$$

  then

  $$\mathrm{signedDigit\_to\_nthDigit}\ r\ p\ 17$$

  reduces to $-1$ or $0$ or $1$
  and not to something like

  $$\mathrm{axiom1}\ (\mathrm{axiom2}\ 5)\ 6$$

- For this we need to make sure that from a postulated axioms we cannot extract any computational content.
- What we want is that if we derive

  $$a : A$$

  where $A$ algebraic data type, $a$ is closed, then $a$ is canonical, i.e. starts with a constructor.

# Restrictions on Postulates (PhD thesis Chi Ming Chuang)

- Postulated functions have as result type equalities or postulated types.
  - Especially negation is not allowed as conclusion because of $\neg A = A \rightarrow \bot$.
- Functions defined by case distinction on equalities have as result type only equalities or postulated types.
  - So when using postulated functions and equalities we stay within equalities and postulated types.

# Equalities

- The problem with equalities was that they occur in conclusions in Agda.
- If we had 2 equalities:
  - one on postulated types,
  - one on non-postulated types,

  then only a restriction on the equality on postulated types is needed.

# Results of PhD Thesis Chi Ming Chuang

- Chi Ming Chuang: Extraction of Programs for Exact Real Number Computation using Agda. PhD thesis, Dept. of Computer Science, Swansea, March 2011
- Chi Ming Chuang
  - showed that under these conditions all closed elements algebraic types are canonical,
  - introduced the signed digit real numbers and showed that they are closed under $\mathrm{av}$, $*$ and contain the rationals,
  - transformed them into programs computing those operations on Reals given by streams of signed digits,
  - was able to execute the resulting programs using a compiled version of Agda.

# Codata in Functional Programming

- $\text{SignedDigit}$ above was defined by a codata type.
- Consider a simpler example:

$$\begin{array}{l} \text{codata Stream} : \text{Set where} \\ \quad \text{cons} : \mathbb{N} \to \text{Stream} \to \text{Stream} \end{array}$$

Codata contains objects such as

$$\text{cons}\ 0\ (\text{cons}\ 0\ (\text{cons}\ 0\ \cdots))$$

- We immediately get non-normalisation.
- Restrictions were applied in Coq and Agda on reductions of elements of codata types.
  - In Coq resulted in problem of subject reduction.
  - In Agda restrictions make codata type not very useful.

# Coalgebras

- Solution is to use approach from category theory.
- Treat coalgebras as we treat functions in the $\lambda$-calculus:
  - There functions are not a set of pairs
    – and therefore an infinite object,
  - but a program which applied to its arguments computes the result.
- Similarly elements of coalgebras are not per se infinite objects, but objects which can be unfolded computationally possibly infinitely often:

$$\begin{array}{l}\mathrm{coalg\ Stream : Set\ where} \\ \quad \begin{array}{lcl}\mathrm{head} & : & \mathrm{Stream} \to \mathbb{N} \\ \mathrm{tail} & : & \mathrm{Stream} \to \mathrm{Stream}\end{array}\end{array}$$

- Idea is: an element of $\mathrm{Stream}$ is any object, to which we can apply $\mathrm{head}$ and $\mathrm{tail}$ and obtain natural numbers or Streams.

# Introduction Rule

$$\begin{array}{l}\mathrm{coalg\ Stream} : \mathrm{Set\ where} \\ \quad \begin{array}{lcl} \mathrm{head} & : & \mathrm{Stream} \to \mathbb{N} \\ \mathrm{tail} & : & \mathrm{Stream} \to \mathrm{Stream} \end{array}\end{array}$$

- Elimination rule for $\mathrm{Stream}$ is given by it's eliminators $\mathrm{head}$, $\mathrm{tail}$.
- Introduction rule is "derived" (not in a mathematical sense) from the principle that elements of $\mathrm{Stream}$ are anything admitting $\mathrm{head}$ and $\mathrm{tail}$.
- Example:

$$\begin{array}{l}\mathrm{inc} : \mathbb{N} \to \mathrm{Stream} \\ \begin{array}{lcl} \mathrm{head}\ (\mathrm{inc}\ n) & = & n \\ \mathrm{tail}\ \ (\mathrm{inc}\ n) & = & \mathrm{inc}\ (n+1) \end{array}\end{array}$$

# Introduction Rules for Coalgebras

- In its simple form (coiteration) elimination rules correspond exactly to the categorical diagram of a weakly final coalgebra.
- More advanced forms (e.g. corecursion) can be derived for final coalgebras and then used to extend weakly final coalgebras.

# Patterns and Copatterns

- In our POPL 2013 paper
  - Andreas Abel, Brigitte Pientka, David Thibodeau and Anton Setzer: Copatterns: programming infinite structures by observations. POPL 2013, pp. 27 - 38

  we
  - showed how to mix pattern and copattern matching, and nest them as well,
  - introduced a small (non-normalising) calculus for mixed and nested pattern and copattern matching,
  - showed that this guarantees that all function definitions are coverage complete,
  - showed that the resulting calculus fulfils subject reduction.

# Example of Patterns and Copatterns

Definition of the stream:
$f\ n = n, n, n{-}1, n{-}1, \ldots 0, 0, N, N, N{-}1, N{-}1, \ldots 0, 0, N, N, N{-}1, N{-}1,$

$$
\begin{array}{llll}
f : \mathbb{N} \to \mathrm{Stream} & & & \\
\mathrm{head} & (f\ 0) & = & 0 \\
\mathrm{head}\ (\mathrm{tail}\ (f\ 0)) & & = & 0 \\
\mathrm{tail}\ (\mathrm{tail}\ (f\ 0)) & & = & f\ N \\
\mathrm{head} & (f\ (\mathrm{S}\ n)) & = & \mathrm{S}\ n \\
\mathrm{head}\ (\mathrm{tail}\ (f\ (\mathrm{S}\ n))) & & = & \mathrm{S}\ n \\
\mathrm{tail}\ (\mathrm{tail}\ (f\ (\mathrm{S}\ n))) & & = & f\ n
\end{array}
$$

- There is an easy algorithm to reduce these definitions back to case distinction operators and full recursion.
- One can trace back the recursion and in some cases reduce it to the primitive (co)recursion operators.

# Berger's Data Type of Continuous Functions

Let $\mathrm{I}$ denote $[-1, 1]$.
Let $\mathrm{I}^{\mathrm{I}}$ be $\mathrm{I} \to \mathrm{I}$ for a postulated $\mathrm{I}$ or postulated type with suitable axioms.

$$\begin{array}{l} \mathrm{coalg\ Cont}\ (f : \mathrm{I}^{\mathrm{I}})\ : \mathrm{Set\ where} \\ \quad \mathrm{elim} : (f : \mathrm{I}^{\mathrm{I}}) \to \mathrm{Cont}\ f \to \mathrm{Contaux}\ f \end{array}$$

$$\begin{array}{l} \mathrm{data\ Contaux}\ (f : \mathrm{I}^{\mathrm{I}}) : \mathrm{Set\ where} \\ \quad \mathrm{consume} \quad : \quad (f : \mathrm{I}^{\mathrm{I}}) \to ((d : \mathrm{Digit}) \to \mathrm{Contaux}(f \circ \mathrm{e}_d)) \to \mathrm{Contaux}\ f \\ \quad \mathrm{produce} \quad : \quad (f : \mathrm{I}^{\mathrm{I}}) \to (d : \mathrm{Digit}) \to \mathrm{Cont}(\mathrm{e}_d^{-1} \circ f) \to \mathrm{Contaux}\ f \end{array}$$

# Conclusion

- Use of postulated Real numbers very good approach to treating real numbers in type theory.
- Restrictions on postulates guarantee that program extraction works.
- Copattern matching is the correct dual of pattern matching.
- Definition of functions on data and codata types by pattern/copattern matching works well.

# Patterns and Copatterns

- We demonstrate this by an example:
- Example define stream:
  $f\ n =$
  $n, n, n-1, n-1, \ldots 0, 0, N, N, N-1, N-1, \ldots 0, 0, N, N, N-1, N-1,$

# Patterns and Copatterns

$$f\ n = n, n, n-1, n-1, \ldots 0, 0, N, N, N-1, N-1, \ldots 0, 0, N, N, N-1, N-1,$$

$$\begin{array}{l} f : \mathbb{N} \to \mathrm{Stream} \\ f \quad = \quad ? \end{array}$$

# Patterns and Copatterns

$f\ n = n, n, n-1, n-1, \ldots 0, 0, N, N, N-1, N-1, \ldots 0, 0, N, N, N-1, N-1,$

$$\begin{array}{l} f : \mathbb{N} \to \mathrm{Stream} \\ f \quad = \quad ? \end{array}$$

Pattern match on $f : \mathbb{N} \to \mathrm{Stream}$:

$$\begin{array}{l} f : \mathbb{N} \to \mathrm{Stream} \\ f\ n \quad = \quad ? \end{array}$$

# Patterns and Copatterns

$f\ n = n, n, n-1, n-1, \ldots 0, 0, N, N, N-1, N-1, \ldots 0, 0, N, N, N-1, N-1,$

$$
\begin{array}{lcl}
f : \mathbb{N} \to \mathrm{Stream} & & \\
f\ n & = & ?
\end{array}
$$

**Copattern matching** on $f\ n : \mathrm{Stream}$:

$$
\begin{array}{lcl}
f : \mathbb{N} \to \mathrm{Stream} & & \\
\mathrm{head}\ (f\ n) & = & ? \\
\mathrm{tail}\ \ (f\ n) & = & ?
\end{array}
$$

# Patterns and Copatterns

$f\ n = n, n, n-1, n-1, \ldots 0, 0, N, N, N-1, N-1, \ldots 0, 0, N, N, N-1, N-1,$

$$\begin{array}{lcl} f : \mathbb{N} \to \mathrm{Stream} & & \\ \mathrm{head}\ (f\ n) & = & ? \\ \mathrm{tail}\ \ (f\ n) & = & ? \end{array}$$

**Pattern matching** on the first $n : \mathbb{N}$:

$$\begin{array}{lcl} f : \mathbb{N} \to \mathrm{Stream} & & \\ \mathrm{head}\ (f\ 0) & = & ? \\ \mathrm{head}\ (f\ (\mathrm{S}\ n)) & = & ? \\ \mathrm{tail}\ \ (f\ n) & = & ? \end{array}$$

# Patterns and Copatterns

$f\ n = n, n, n{-}1, n{-}1, \ldots 0, 0, N, N, N{-}1, N{-}1, \ldots 0, 0, N, N, N{-}1, N{-}1,$

$$
\begin{array}{lcl}
f : \mathbb{N} \to \mathrm{Stream} & & \\
\mathrm{head}\ (f\ 0) & = & ? \\
\mathrm{head}\ (f\ (\mathrm{S}\ n)) & = & ? \\
\mathrm{tail}\ \ (f\ n) & = & ?
\end{array}
$$

**Pattern matching** on second $n : \mathbb{N}$:

$$
\begin{array}{lcl}
f : \mathbb{N} \to \mathrm{Stream} & & \\
\mathrm{head}\ (f\ 0) & = & ? \\
\mathrm{head}\ (f\ (\mathrm{S}\ n)) & = & ? \\
\mathrm{tail}\ \ (f\ 0) & = & ? \\
\mathrm{tail}\ \ (f\ (\mathrm{S}\ n)) & = & ?
\end{array}
$$

# Patterns and Copatterns

$f\ n = n, n, n-1, n-1, \ldots 0, 0, N, N, N-1, N-1, \ldots 0, 0, N, N, N-1, N-1,$

$$
\begin{array}{lcl}
f : \mathbb{N} \to \mathrm{Stream} & & \\
\mathrm{head}\ (f\ 0) & = & ? \\
\mathrm{head}\ (f\ (\mathrm{S}\ n)) & = & ? \\
\mathrm{tail}\ \ (f\ 0) & = & ? \\
\mathrm{tail}\ \ (f\ (\mathrm{S}\ n)) & = & ?
\end{array}
$$

**Copattern matching** on $\mathrm{tail}\ (f\ 0) : \mathrm{Stream}$

$$
\begin{array}{lcl}
f : \mathbb{N} \to \mathrm{Stream} & & \\
\mathrm{head}\ (f\ 0) & = & ? \\
\mathrm{head}\ (f\ (\mathrm{S}\ n)) & = & ? \\
\mathrm{head}\ (\mathrm{tail}\ (f\ 0)) & = & ? \\
\mathrm{tail}\ (\mathrm{tail}\ (f\ 0)) & = & ? \\
\mathrm{tail}\ (f\ (\mathrm{S}\ n)) & = & ?
\end{array}
$$

# Patterns and Copatterns

$$
\begin{array}{l}
f : \mathbb{N} \rightarrow \text{Stream} \\
\text{head} \quad (f\ 0 \quad ) = \quad ? \\
\text{head} \quad (f\ (\text{S} \quad n)) = \quad ? \\
\text{head}\ (\text{tail}\ (f\ 0 \quad )) = \quad ? \\
\text{tail} \quad (\text{tail}\ (f\ 0 \quad )) = \quad ? \\
\text{tail} \quad (f\ (\text{S}\ n \ )) = \quad ?
\end{array}
$$

**Copattern matching** on $\text{tail}\ (f\ (\text{S}\ n)) : \text{Stream}$:

$$
\begin{array}{l}
f : \mathbb{N} \rightarrow \text{Stream} \\
\text{head} \quad (f\ 0 \quad ) = \quad ? \\
\text{head} \quad (f\ (\text{S}\ n)) = \quad ? \\
\text{head}\ (\text{tail}\ (f\ 0 \quad )) = \quad ? \\
\text{tail} \quad (\text{tail}\ (f\ 0 \quad )) = \quad ? \\
\text{head}\ (\text{tail}\ (f\ (\text{S}\ n))) = \quad ? \\
\text{tail} \quad (\text{tail}\ (f\ (\text{S}\ n))) = \quad ?
\end{array}
$$

# Patterns and Copatterns

We resolve the goals:

$$
\begin{array}{llll}
f : \mathbb{N} \to \mathrm{Stream} & & & \\
\mathrm{head} & (f\ 0\ \ ) & = & 0 \\
\mathrm{head}\ (\mathrm{tail} & (f\ 0\ \ )) & = & 0 \\
\mathrm{tail}\ \ (\mathrm{tail} & (f\ 0\ \ )) & = & f\ N \\
\mathrm{head} & (f\ (\mathrm{S}\ n)) & = & \mathrm{S}\ n \\
\mathrm{head}\ (\mathrm{tail} & (f\ (\mathrm{S}\ n))) & = & \mathrm{S}\ n \\
\mathrm{tail}\ \ (\mathrm{tail} & (f\ (\mathrm{S}\ n))) & = & f\ n
\end{array}
$$

- There is an easy algorithm to reduce these definitions back to case distinction operators and full recursion.
- One can trace back the recursion and in some cases reduce it to the primitive (co)recursion operators.

# Multiple Constructors in Algebras and Coalgebras

- Having more than one constructor in algebras correspond to disjoint union:

$$\begin{array}{l} \text{data } \mathbb{N} : \text{Set where} \\ \quad \begin{array}{lcl} 0 & : & \mathbb{N} \\ \text{S} & : & \mathbb{N} \to \mathbb{N} \end{array} \end{array}$$

  corresponds to

$$\begin{array}{l} \text{data } \mathbb{N} : \text{Set where} \\ \quad \begin{array}{lcl} \text{intro} & : & (1 + \mathbb{N}) \to \mathbb{N} \end{array} \end{array}$$

# Multiple Constructors in Algebras and Coalgebras

- Dual of disjoint union is products, and therefore multiple destructors correspond to product:

$$\begin{array}{l}\text{coalg Stream} : \text{Set where} \\ \quad \begin{array}{lcl}\text{head} & : & \text{Stream} \rightarrow \mathbb{N} \\ \text{tail} & : & \text{Stream} \rightarrow \text{Stream}\end{array}\end{array}$$

corresponds to

$$\begin{array}{l}\text{coalg Stream} : \text{Set where} \\ \quad \begin{array}{lcl}\text{case} & : & \text{Stream} \rightarrow (\mathbb{N} \times \text{Stream})\end{array}\end{array}$$

# Codata Types Correspond to Disjoint Union

- Consider

$$\begin{array}{l} \text{codata coList} : \text{Set where} \\ \quad \begin{array}{lcl} \text{nil} & : & \text{coList} \\ \text{cons} & : & \mathbb{N} \to \text{coList} \to \text{coList} \end{array} \end{array}$$

- Cannot be simulated by using several destructors.

# Simulating Codata Types by Simultaneous Algebras/Coalgebras

- Represent Codata as follows

mutual

  coalg coList : Set where
    unfold : coList → coListShape

  data coListShape : Set where
    nil   :  coListShape
    cons  :  $\mathbb{N}$ → coList → coListShape

# Definition of Append

$$\text{append} : \text{coList} \to \text{coList} \to \text{coList}$$
$$\text{append}\ l\ l' = ?$$

# Definition of Append

$$\begin{array}{l} \text{append} : \text{coList} \to \text{coList} \to \text{coList} \\ \text{append}\ l\ l' = ? \end{array}$$

We copattern match on $\text{append}\ l\ l' : \text{coList}$:

$$\begin{array}{l} \text{append} : \text{coList} \to \text{coList} \to \text{coList} \\ \text{unfold}\ (\text{append}\ l\ l') = ? \end{array}$$

# Definition of Append

$$\begin{array}{l} \text{append} : \text{coList} \to \text{coList} \to \text{coList} \\ \text{unfold}\ (\text{append}\ l\ l') = ? \end{array}$$

We cannot pattern match on $l$.
But we can do so on (unfold $l$):

$$\begin{array}{l} \text{append} : \text{coList} \to \text{coList} \to \text{coList} \\ \text{unfold}\ (\text{append}\ l\ l') = \\ \quad \text{case}\ (\text{unfold}\ l)\ \text{of} \\ \qquad \begin{array}{lcl} \text{nil} & \to & ? \\ (\text{cons}\ n\ l) & \to & ? \end{array} \end{array}$$

# Definition of Append

$$
\begin{array}{l}
\text{append} : \text{coList} \to \text{coList} \to \text{coList} \\
\text{unfold}\ (\text{append}\ l\ l') = \\
\quad \text{case}\ (\text{unfold}\ l)\ \text{of} \\
\quad\quad \begin{array}{lcl} \text{nil} & \to & ? \\ (\text{cons}\ n\ l) & \to & ? \end{array}
\end{array}
$$

We resolve the goals:

$$
\begin{array}{l}
\text{append} : \text{coList} \to \text{coList} \to \text{coList} \\
\text{unfold}\ (\text{append}\ l\ l') = \\
\quad \text{case}\ (\text{unfold}\ l)\ \text{of} \\
\quad\quad \begin{array}{lcl} \text{nil} & \to & \text{unfold}\ l' \\ (\text{cons}\ n\ l) & \to & \text{cons}\ n\ (\text{append}\ l\ l') \end{array}
\end{array}
$$