Programming with Objects in Theorem Provers based on Martin Löf Type Theory

Anton Setzer
Swansea University, Swansea UK
(Joint work with Andreas Abel and Stephan Adelsberger)
Talk given at Workshop on Mathematical Logic and its Application, Kyoto University, Japan

16 September 2016

Agda

- Agda is a theorem prover based on Martin-Löf's intuitionistic type theory.
- Based on Propositions of types
  - For A a data type $a : A$ means a is an element of A
  - For A proposition $a : A$ means a is a proof of A.
- Programs are defined recursively.
- Termination checker guarantees all program terminate.
  Otherwise Agda would be inconsistent:

```
q : ⊥
q = q
```

- For historic reasons **types** denoted by keyword Set.
- There are as well higher type levels

$$\mathrm{Set} \overset{\subseteq}{:} \mathrm{Set}_1 \overset{\subseteq}{:} \mathrm{Set}_2 \overset{\subseteq}{:} \cdots$$

Dependent Function Types

- Main type forming constructs in Agda are
  - dependent function types,
  - algebraic data types,
  - record types.
- The dependent function type

$$(x \ : \ A) \ \rightarrow C \, x$$

  is the type of functions mapping $a : A$ to an element of type $C \ a$.
- E.g.

$$\mathrm{matmult} : (\ n \ m \ k : \mathbb{N} \ ) \rightarrow \mathrm{Mat} \ n \ m \rightarrow \mathrm{Mat} \ m \ k \rightarrow \mathrm{Mat} \ n \ k$$

Algebraic data types

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

```
g : ℕ → ℕ
g 0             = 5
g (suc 0)       = 12
g (suc (suc n)) = g n * n
```

Syntax in Agda

- Agda allows hidden arguments

$$\text{cons} : \{X : \text{Set}\} \to X \to \text{List } X \to \text{List } X$$

$$\text{l} : \text{List } \mathbb{N}$$
$$\text{l} = \text{cons } 0 \text{ nil}$$

$$\text{l'} : \text{List } \mathbb{N}$$
$$\text{l'} = \text{cons } \{\mathbb{N}\} \; 0 \text{ nil}$$

- Agda has mixfix symbols. Syntax example

$$\text{if\_then\_else} : \{X : \text{Set}\} \to \text{Bool} \to X \to X \to X$$
$$\text{if true then } x \text{ else } y = x$$
$$\text{if false then } x \text{ else } y = y$$

Solution: Coalgebras Defined by Observations

- We define coalgebras by their observations. Tentative syntax

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

- Stream is the largest set of terms which allow arbitrary many applications of tail followed by head to obtain a natural numbers.
- Therefore no infinite expansion of streams:
  – for each expansion of a stream one needs one application of tail.

Principle of Guarded Recursion

- Define

$$
\begin{array}{llcll}
f : A \to \mathrm{Stream} & & & & \\
\mathrm{head} & (f \;\; a) & = & \cdots & : \mathbb{N} \\
\mathrm{tail} & (f \;\; a) & = & \cdots & : \mathrm{Stream}
\end{array}
$$

  where

$$
\begin{array}{lcllcl}
\mathrm{tail}\,(f\;\,a) & = & f\;a' & \text{for some} & a' & : A \\
\text{or} & & & & & \\
\mathrm{tail}\,(f\;\,a) & = & s' & \text{for some} & s' & : \mathrm{Stream} \text{ given before}
\end{array}
$$

- **No** function can be applied to the corecursion hypothesis.
- Using sized types one can apply size preserving or size increasing functions to co-IH (Abel).
- Above is example of **copattern matching**.

Example

- Constant stream of $a, a, a, \ldots$

$$\begin{array}{lcl} \mathsf{const} : \{A : \mathsf{Set}\} \to A \to \mathsf{Stream}\ \ A & & \\ \mathsf{head}\ (\mathsf{const}\ a) & = & a \\ \mathsf{tail}\ \ (\mathsf{const}\ a) & = & \mathsf{const}\ \ a \end{array}$$

- The increasing stream $n, n+1, n+2, \ldots$

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

- Cons is **defined**:

$$\begin{array}{lcl} \mathsf{cons}\ \ : X \to \mathsf{Stream}\ \ X \to \mathsf{Stream}\ \ X & & \\ \mathsf{head}\ \ \ (\mathsf{cons}\ \ x\ l) & = & x \\ \mathsf{tail}\ \ \ \ (\mathsf{cons}\ \ x\ l) & = & l \end{array}$$

Syntax in Agda

- In Agda the record type has been reused for defining coalgebras:

```
record Stream (A : Set) : Set where
  coinductive
  constructor _::_
  field
    head  :  A
    tail  :  Stream A
```

const and inc can be defined with the syntax as given before

Nested Patter/Copattern Matching

- We can even define functions by a combination of pattern and copattern matching and nest those:
  The following defines the stream

$$\text{stutterDown}\ \ n\ n \ \ = \ \ n, n, n-1, n-1, \ldots 0, 0, n, n, n-1, n-1, \ldots$$

$$
\begin{array}{ll}
\text{stutterDown} : \mathbb{N} \to \mathbb{N} \to \text{Stream}\ \mathbb{N} & \\
\text{head}\ (\text{stutterDown}\ n\ m) & = m \\
\text{head}\ (\text{tail}\ (\text{stutterDown}\ n\ m)) & = m \\
\text{tail}\ (\text{tail}\ (\text{stutterDown}\ n\ (\text{suc}\ m))) & = \text{stutterDown}\ n\ m \\
\text{tail}\ (\text{tail}\ (\text{stutterDown}\ n\ 0)) & = \text{stutterDown}\ n\ n
\end{array}
$$

Object-Oriented/Based Programming

- Object-oriented (OO) programming is currently main programming paradigm.
- Good for bundling operations into one objects, hiding implementations and reuse of code.
- Here restriction to **object-based programming**.
  - Only notion of an object covered.
- Ultimate goal: use objects in order to organise proofs in a better way.

Example: cell in Java

```
class cell <A> {

    /* Instance Variable */
    A content;

    /* Constructor */
    cell (A s) { content = s; }

    /* Method put */
    public void put (A s) { content = s; }

    /* Method get */
    public A get () { return content; }
}
```

Modelling Methods as Objects

- The Type (interface) cell modelled as a coalgebra Cell.
- A method

$$B \ \mathrm{m} \ (A \ x)$$

  is modelled as observation

$$\mathsf{m} : \mathsf{Cell} \to \mathsf{A} \to \mathsf{B} \times \mathsf{Cell}$$

- Return type void is modelled as Unit (one element type).
- A constructor with argument A modelled as a function defined by guarded recursion

$$\mathsf{cell} : \mathsf{A} \to \mathsf{Cell}$$

Object as a Coalgebra

Using coalg notation we obtain

```
coalg Cell (A : Set) where
    put  :  Cell A  →  A     →  (Unit × Cell A)
    get  :  Cell A  →  Unit  →  (A    × Cell A)

cell : {A : Set} → A → Cell A
put   (cell a)  b  =  (unit  ,  cell b)
get   (cell a)  _  =  (a     ,  cell a)
```

Official Agda Code

```
record Cell (X : Set) : Set where
  coinductive
  field
    put : X    → Unit × Cell X
    get : Unit → X    × Cell X

cell : {X : Set} → X → Cell X
put (cell x) y = (unit , cell y)
get (cell x) _ = (x    , cell x)
```

Generic Version

An interface for an object consist of methods and the result type:

```
record Interface   :  Set₁ where
  field  Method    :  Set
         Result    :  Method → Set
```

An Object of an interface I has a method which for every method returns an element of the result type and the updated object:

```
record Object (I : Interface) : Set where
  coinductive
  field objectMethod : (m : Method I) → Result I m × Object I
```

State Dependent Interface

```
record Interfaceˢ : Set₁ where
  field
    Stateˢ  : Set
    Methodˢ : Stateˢ → Set
    Resultˢ : (s : Stateˢ) → (m : Methodˢ s) → Set
    nextˢ   : (s : Stateˢ) → (m : Methodˢ s) → Resultˢ s m
              → Stateˢ
```

State Dependent Object

Assuming I : $\text{Interface}^{\text{s}}$ we define the set of state dependent objects:

$$
\begin{array}{l}
\text{record } \text{Object}^{\text{s}}\ (I : \text{Interface}^{\text{s}})\ (s : \text{State}^{\text{s}}\ I) : \text{Set where} \\
\text{coinductive} \\
\text{field} \\
\quad \text{objectMethod} : \ (m : \text{Method}^{\text{s}}\ I\ s) \\
\qquad\qquad\qquad\quad \to \Sigma[\ r \in \text{Result}^{\text{s}}\ I\ s\ m\ ]\ \text{Object}^{\text{s}}\ I\ (\text{next}^{\text{s}}\ I\ s\ m\ r)
\end{array}
$$

Example Safe Stack

$\text{StackState}^{\text{s}} = \mathbb{N}$

$$\begin{array}{l}
\text{data StackMethod}^{\text{s}}\ (A : \text{Set}) : \text{StackState}^{\text{s}} \to \text{Set where} \\
\quad \text{push} : \{n : \text{StackState}^{\text{s}}\} \to A \to \text{StackMethod}^{\text{s}}\ A\ n \\
\quad \text{pop} \ \ : \{n : \text{StackState}^{\text{s}}\} \to \text{StackMethod}^{\text{s}}\ A\ (\text{suc}\ n)
\end{array}$$

$$\begin{array}{l}
\text{StackResult}^{\text{s}} : (A : \text{Set}) \to (s : \text{StackState}^{\text{s}}) \to \text{StackMethod}^{\text{s}}\ A\ s \\
\qquad\qquad\qquad \to \text{Set} \\
\text{StackResult}^{\text{s}}\ A\ .n\ (\text{push}\ \{\ n\ \}\ x_1) = \text{Unit} \\
\text{StackResult}^{\text{s}}\ A\ (\text{suc}\ .n)\ (\text{pop}\ \{n\}\ ) = A
\end{array}$$

$$\begin{array}{l}
\text{n}^{\text{s}} : (A : \text{Set}) \to (s : \text{StackState}^{\text{s}}) \to (m : \text{StackMethod}^{\text{s}}\ A\ s) \\
\qquad \to (r : \text{StackResult}^{\text{s}}\ A\ s\ m) \to \text{StackState}^{\text{s}} \\
\text{n}^{\text{s}}\ A\ .n\ (\text{push}\ \{\ n\ \}\ x)\ \ r = \text{suc}\ n \\
\text{n}^{\text{s}}\ A\ (\text{suc}\ .n)\ (\text{pop}\ \{\ n\ \})\ r = n
\end{array}$$

Safe Stack

$$
\begin{array}{llll}
\mathsf{StackInterface}^{s} : (A : \mathsf{Set}) \to \mathsf{Interface}^{s} \\
\mathsf{State}^{s} & (\mathsf{StackInterface}^{s}\ A) & = & \mathsf{StackState}^{s} \\
\mathsf{Method}^{s} & (\mathsf{StackInterface}^{s}\ A) & = & \mathsf{StackMethod}^{s}\ A \\
\mathsf{Result}^{s} & (\mathsf{StackInterface}^{s}\ A) & = & \mathsf{StackResult}^{s}\ A \\
\mathsf{next}^{s} & (\mathsf{StackInterface}^{s}\ A) & = & \mathsf{n}^{s}\ A
\end{array}
$$

$$
\begin{array}{l}
\mathsf{stackO} : \forall \{E : \mathsf{Set}\}\ \{n : \mathbb{N}\}\ (v : \mathsf{Vec}\ E\ n) \\
\qquad \to \mathsf{Object}^{s}\ (\mathsf{StackInterface}^{s}\ E)\ n \\
\mathsf{objectMethod}\ (\mathsf{stackO}\ es)\ (\mathsf{push}\ e) = (\_\ ,\ \mathsf{stackO}\ (e :: es)) \\
\mathsf{objectMethod}\ (\mathsf{stackO}\ (e :: es))\ \mathsf{pop} = (e\ ,\ \mathsf{stackO}\ es)
\end{array}
$$

Example Fibonacci Stack

```
data FibState : Set where
  fib : ℕ → FibState
  val : ℕ → FibState


data FibStackEl : Set where
  _+·    : ℕ → FibStackEl
  ·+fib_ : ℕ → FibStackEl

FibStack : ℕ → Set
FibStack = Objectˢ (StackInterfaceˢ FibStackEl)

emptyFibStack : FibStack 0
emptyFibStack = stackO []
```

Reduce

```
reduce : Stackmachine → Stackmachine ⊎ ℕ
reduce (n , fib 0 , stack) = inj₁ (n , val 1 , stack)
reduce (n , fib 1 , stack) = inj₁ (n , val 1 , stack)
reduce (n , fib (suc (suc m)) , stack) =
   objectMethod stack (push (·+fib m)) ▷ λ { (_ , stack₁) →
   inj₁ ( suc n , fib (suc m) , stack₁)        }
reduce (0 , val m , stack) = inj₂ m
reduce (suc n , val m , stack) =
   objectMethod stack pop ▷            λ {  (k +· , stack₁) →
   inj₁ (n , val (k + m) , stack₁) ;
                                              (·+fib k , stack₁) →
   objectMethod stack₁ (push (m +·)) ▷ λ {(_ , stack₂) →
   inj₁ (suc n , fib k , stack₂) }  }
```

Fibonacci Function

```
{-# NON_TERMINATING #-}
iter : Stackmachine → ℕ
iter stack  with reduce stack
... | inj₁ s'  = iter s'
... | inj₂ m   = m

fibUsingStack : ℕ → ℕ
fibUsingStack n = iter (0 , fib n , emptyFibStack)
```

Conclusion

- Definition of coinductive data types (coalgebras) by their observations.
  - Use of **copattern** matching
- Objects as examples of coalgebras.
- State dependent objects.
- Future work
  - Define Gray codes using objects
    - Asymmetry between constructors and observations.
  - Use of objects in organising proofs.
- Use of coalgebras for defining processes: See talk by Bashar Igried at TyDe'2016 in Nara.

Bibliography I

Andreas Abel, Stephan Adelsberger, and Anton Setzer.
Interactive programming in Agda – objects and graphical user interfaces.
To appear in Journal of Functional Programming. Preprint available at http://www.cs.swan.ac.uk/∼csetzer/articles/ooAgda.pdf, 2016.

Bashar Igried and Anton Setzer.
Programming with monadic CSP-style processes in dependent type theory.
To appear in proceedings of TyDe 2016, Type-driven Development, preprint available from
http://www.cs.swan.ac.uk/∼csetzer/articles/TyDe2016.pdf, 2016.

Bibliography II

Anton Setzer.
Object-oriented programming in dependent type theory.
In Conference Proceedings of TFP 2006, 2006.
Available from
http://www.cs.nott.ac.uk/∼nhn/TFP2006/TFP2006-Programme.html
and http://www.cs.swan.ac.uk/∼csetzer/index.html.

Anto Setzer.
How to reason coinductively informally.
In Reinhard Kahle, Thomas Strahm, and Thomas Studer, editors,
Advances in Proof Theory, pages 377–408. Springer, 2016.