MiniAgda: a toy language for integrating dependent and sized types

Edoardo Putti, Lorena Yunes Arriaga

June 6, 2019

Definitions

Definition

Corecursion is a type of operation that is dual to recursion. Whereas recursion works analytically, starting on data further from a base case and breaking it down into smaller data and repeating until one reaches a base case, corecursion works synthetically, starting from a base case and building it up, iteratively producing data further removed from a base case

Definition

Corecursive function: A function which calls itself in the implementation

Definition

Codata: is a reference to a computation that, when executed, may produce (amongst other things) more codata.

Definition

Guarded Condition The recursive call is directly under the coconstructor.

• *Coinductive types* admit potentially infinite inhabitants, the most basic and most prominent example being streams. They are the basic coinductive datatype on which we define corecursive functions.

• In the context of coinductive types, corecursive definitions require productivity.

Definition

Productivity: The next finite amount of information of a stream must always be available in a finite amount of time.

•A simple criterion for productivity which can be checked syntactically is guardedness

• Using *sized coinductive types* we can keep track in the type system whether a function is stream destructing, stream constructing or depth preserving.

• Thus, sized types can offer a systematic solution to the *"guardedness-mediated-by-functions"* problem.

• Sized coinductive type definitions look very similar to their inductive counterpart: The rules to annotate the recursive occurrences of the coinductive type in the types of the coconstructors are identical to the rules for sized constructors.

Definition

The size i of a Stream is a lower bound on the number of coconstructors, called depth of the stream.

• A fully constructed stream will always have size ∞ , but during the construction of the stream we reason with approximations, i.e., streams which have depth i for some arbitrary i.

Depth preserving functions

We can define map as a depth-preserving corecursive function on streams:

```
cofun map : [A : Set] -> [B : Set] -> [i : Size] ->
            (A -> B) -> Stream A i -> Stream B i
{
    map A B ($ i) f (cons .i x xs) = cons i (f x) (map A B i f xs)
}
```

Depth preserving functions

```
fun leq : Nat -> Nat -> [C : Set] -> C -> C -> C {
        leq zero      y         C t f = t;
        leq (succ x)  zero      C t f = f;
        leq (succ x)  (succ y) C t f = leq x y C t f
}

cofun merge: [i : Size] -> Stream Nat i -> Stream Nat i -> Stream Nat i {
        merge ($ i) (cons .i x xs) (cons .i y ys) =
                leq x y (Stream Nat ($ i))
                        (cons i x (merge i xs (cons i y ys)))
                        (cons i y (merge i (cons i x xs) ys))
}
```

The more precise information that its depth is the sum of the depths of the input streams is not expressible in the size language of MiniAgda

Where guarded corecursion fails sized types survive!

```
cofun ham : [i : Size] -> Stream Nat i {
        ham ($ i) = cons i (succ zero)
                    (merge i (map Nat Nat i double (ham i))
                             (map Nat Nat i triple (ham i)))
}
```

Coalgebra and bisimilarity

Definition

Streams over X are infinite sequences of elements from X, called carrier of the stream system. We denote the set of streams over X as the set X^ω.
Given $x \in X$ we use x^* as a shorthand for the stream of all x.

Definition

Derivative of a stream AKA tail
Given a stream δ we denote with δ^0 the first element and with δ' the tail

Definition

Bisimilarity AKA behavioural equivalence
Two streams δ and τ are bisimilar ($\delta \sim \tau$) if $\delta^0 = \tau^0$ and $\delta' \sim \tau'$

Add dependent types and stir for 10 minutes

We can encode predicates such as bisimilarity in type theory by using dependent types.

```
sized codata StreamEq (A : Set) : (i : Size) -> Stream A i -> Stream A i -> Set {
        bisim : [i : Size] -> [a : A] -> [as : Stream A i] -> [bs : Stream A i] ->
                StreamEq A    i              as              bs ->
                StreamEq A ($ i) (cons i a as) (cons i a bs);
}
```

Reflexivity

Lemma

In a stream system the bisimilarity relation is reflexive

```
-- x ∼ x
cofun reflexivity : [A : Set] -> [i : Size] -> (s : Stream A i) ->
        StreamEq A i s s
{
        reflexivity A ($ i) (cons .i a as) = bisim i a as as (reflexivity A i as)
}
```

Symmetry

Lemma

In a stream system the bisimilarity relation is symmetric

```
-- x ∼ y  then  y ∼ x
cofun symmetry : [A : Set] -> [i : Size] -> (x : Stream A i) -> (y : Stream A i) ->
         -- x ∼ y
         StreamEq A i x y ->
         -- y ∼ x
         StreamEq A i y x
{
                                                        -- decomposition of a proof
         symmetry A ($ i) (cons .i .a xs) (cons .i .a ys) (bisim .i a .xs .ys p)
                                       -- reuse of the proof
                  = bisim i a ys xs (symmetry A i xs ys p)
}
```

Transitivity

Lemma

In a stream system the bisimilarity relation is transitive

```
-- prove that x ∼ y and y ∼ z implies x ∼ z
cofun transitivity : [A : Set] -> [i : Size] ->
        (x : Stream A i) -> (y : Stream A i) -> (z : Stream A i) ->
        -- x ∼ y
        StreamEq A i x y ->
        -- y ∼ z
        StreamEq A i y z ->
        -- x ∼ z
        StreamEq A i x z
{
        transitivity A ($ i) (cons .i .a xs) (cons .i .a ys) (cons .i .a zs)
                (bisim .i .a .xs .ys p1)
                (bisim .i a .ys .zs p2)
                = bisim i a xs zs (transitivity A i xs ys zs p1 p2)
}
```

Proving behavioural equivalence of complex streams - 1

Lemma

Let $f : A \to B$ then map $f\ a* \sim (fa)*$

```
-- prove that map f a* ~ (fa)*
cofun map_repeat : [A : Set] -> [B : Set] -> [i : Size] -> (f : A -> B) -> (a : A) -
        StreamEq B i
                (map A B i f (repeat A i a))
                (repeat B i (f a))
{
        map_repeat A B ($ i) f a
        =
        bisim i (f a)
                (map A B i f (repeat A i a))
                (repeat B i (f a))
                (map_repeat A B i f a)
}
```

Proving behavioural equivalence of complex streams - 2

Lemma

Let $f : A \to B$ *and* $g : B \to C$ *then* $map\ (g \circ f)\ a* \sim map\ g\ (map\ f\ a*)$

```
-- prove that map g ∘ f a* ∼ map g (map f a*)
cofun map_composition : [A : Set] -> [B : Set] -> [C : Set] -> [i : Size] ->
        (f : A -> B) -> (g : B -> C) ->
        (a : A) ->
        StreamEq C i
                (map A C i (\ n -> g (f n)) (repeat A i a))
                (map B C i g (map A B i f (repeat A i a)))
{
        map_composition A B C ($ i) f g a =
                bisim i (g (f a))
                        (map A C i (\ n -> (g (f n))) (repeat A i a))
                        (map B C i g (map A B i f (repeat A i a)))
                        (map_composition A B C i f g a)
}
```

Conclusions

Thank you for your attention
Thank to Niels for his help

References

1. Abel, A. (2010). MiniAgda: Integrating sized and dependent types. arXiv preprint arXiv:1012.4896.
2. Abel, A. (2012). Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. arXiv preprint arXiv:1202.3496.
3. Thibodeau, D. Termination Checking: Comparing Structural Recursion and Sized Types by Examples.