Induction, Recursion, Coinduction, Corecursion

Jan van Eijck

June 11, 2002

Abstract

More on induction and recursion. Creating infinite datastructures with corecursion.

```
module RAC8

where

import STAL (display)
```

A Question about `map` and `filter`

The two operations `map` and `filter` can be combined:

```
Prelude> filter (>4) (map (+1) [1..10])
[5,6,7,8,9,10,11]
Prelude> map (+1) (filter (>4)  [1..10])
[6,7,8,9,10,11]
```

These outcomes are different. This is because the test `(>4)` yields a different result after all numbers are increased by 1.

When we make sure that the test used in the filter takes this change into account, we get the same answers:

```
Prelude> filter (>4) (map (+1) [1..10])
[5,6,7,8,9,10,11]
Prelude> map (+1) (filter ((>4).(+1))  [1..10])
[5,6,7,8,9,10,11]
```

Here `(f . g)` denotes the result of first applying `g` and next `f`. Note that `((>4).(+1))` defines the same property as `(>3)`.

Exercise

Show that for every finite list `xs :: [a]`, every function `f :: a -> b`, every total predicate `p :: b -> Bool` the following holds:

$$\text{filter } p \ (\text{map } f \ \text{xs}) = \text{map } f \ (\text{filter } (p.f) \ \text{xs}).$$

Note: a predicate `p :: b -> Bool` is total if for every object `x :: b`, the application `p x` gives either true or false. In particular, for no `x :: b` does `p x` give rise to an error message.

Solution

Proof by induction on `xs` that

$$\text{filter } p \ (\text{map } f \ \text{xs}) = \text{map } f \ (\text{filter } (p.f) \ \text{xs}).$$

Basis:

$$\text{filter } p \ (\text{map } f \ []) = [] = \text{map } f \ (\text{filter } (p.f) \ []).$$

Induction step: Assume

$$\text{filter } p \ (\text{map } f \ \text{xs}) = \text{map } f \ (\text{filter } (p.f) \ \text{xs}).$$

Consider `(x:xs)`. There are two cases:

1. `p (f x) = True`
2. `p (f x) = False`.

In case (1) we have:

$$
\begin{aligned}
\text{filter } p\ (\text{map } f\ (\text{x:xs})) &\overset{\text{map}}{=} \text{filter } p\ (f\ x) : (\text{map } f\ (\text{x:xs})) \\
&\overset{\text{filter}}{=} (f\ x) : (\text{filter } p\ (\text{map } f\ (\text{x:xs}))) \\
&\overset{\text{ih}}{=} (f\ x) : (\text{map } f\ (\text{filter } (p.f)\ \text{xs})) \\
&\overset{\text{map}}{=} \text{map } f\ (x : (\text{filter } (p.f)\ \text{xs})) \\
&\overset{\text{filter}}{=} \text{map } f\ (\text{filter } (p.f)\ (\text{x:xs})).
\end{aligned}
$$

In case (2) we have:

$$
\begin{aligned}
\text{filter } p\ (\text{map } f\ (\text{x:xs})) &\overset{\text{map}}{=} \text{filter } p\ (f\ x) : (\text{map } f\ (\text{x:xs})) \\
&\overset{\text{filter}}{=} \text{filter } p\ (\text{map } f\ (\text{x:xs})) \\
&\overset{\text{ih}}{=} \text{map } f\ (\text{filter } (p.f)\ \text{xs}) \\
&\overset{\text{map}}{=} \text{map } f\ (\text{filter } (p.f)\ \text{xs}) \\
&\overset{\text{filter}}{=} \text{map } f\ (\text{filter } (p.f)\ (\text{x:xs})).
\end{aligned}
$$

The Tower of Hanoi

The Tower of Hanoi is a tower of 8 disks of different sizes, stacked in order of decreasing size on a peg. Next to the tower, there are two more pegs. The task is to transfer the whole stack of disks to one of the other pegs (using the third peg as an auxiliary) while keeping to the following rules: (i) move only one disk at a time, (ii) never place a larger disk on top of a smaller one.

1. How many moves does it take to completely transfer a tower consisting of n disks?
2. Prove by mathematical induction that your answer to the previous question is correct.
3. How many moves does it take to completely transfer the tower of Hanoi?

Exercise

Can you also find a formula for the number of moves of the disk of size k during the transfer of a tower with disks of sizes $1, \ldots, n$, and $1 \leq k \leq n$? Again, you should prove by mathematical induction that your formula is correct.

Solution

Disk k makes exactly 2^{n-k} moves. To prove this with induction, we prove by induction on m that the disk of size $n - m$ makes 2^m moves. From this the result follows, since $k = n - m$ implies $m = n - k$.

Proof by induction on m that the disk of size $n-m$ makes 2^m moves.

Basis: For $m=0$ we get that the disk of size $n=n-0$ makes $2^0=1$ move. This is correct, for the largest disk moves exactly once, from source to destination.

Induction step: Assume that disk $n-m$ makes 2^{n-m} moves. Now there are two kinds of moves for disk $n-(m+1)$: (i) move it on top of disk $n-m$, or (ii) remove it from disk $n-m$. This makes clear that to every single move of disk $n-m$ there are two moves of disk $n-(m+1)$, giving $2\times 2^{n-m}=2^{n-(m+1)}$ moves altogether.

Note that this outcome squares with the formula for the number of moves to shift a complete tower. If disk k makes 2^{n-k} moves the total number of moves is $\sum_{k=1}^{n} 2^{n-k}$. Since $1+\sum_{k=1}^{n} 2^{n-k}=2^n$ (use binary representation to see this) we get $\sum_{k=1}^{n} 2^{n-k}=2^n-1$.

A Tower of Hanoi Program

For an implementation of the disk transfer procedure, an obvious way to represent the starting configuration of the tower of Hanoi is:

`([1,2,3,4,5,6,7,8],[],[])`

For clarity, we give the three pegs names `A`, `B` and `C`. and we declare a type `Tower`:

```
data Peg   = A | B | C
type Tower = ([Int], [Int], [Int])
```

There are six possible single moves from one peg to another:

```
move :: Peg  -> Peg -> Tower -> Tower
move A B (x:xs,ys,zs) =  (xs,x:ys,zs)
move B A (xs,y:ys,zs) =  (y:xs,ys,zs)
move A C (x:xs,ys,zs) =  (xs,ys,x:zs)
move C A (xs,ys,z:zs) =  (z:xs,ys,zs)
move B C (xs,y:ys,zs) =  (xs,ys,y:zs)
move C B (xs,ys,z:zs) =  (xs,z:ys,zs)
```

The procedure `transfer` takes three arguments for the pegs, an argument for the number of disks to move, and an argument for the tower configuration to move. The output is a list of tower configurations.

```
transfer :: Peg -> Peg -> Peg -> Int -> Tower
                                     -> [Tower]
transfer _ _ _ 0 tower = [tower]
transfer p q r n tower = transfer p r q (n-1) tower
                         ++
                         transfer r q p (n-1)
                                  (move p q tower')
   where tower' = last (transfer p r q (n-1) tower)

hanoi :: Int -> [Tower]
hanoi n = transfer A C B n ([1..n],[],[])
```

Here is the output for `hanoi 4`:

```
IAR> hanoi 4
[([1,2,3,4],[],[]),([2,3,4],[1],[]),([3,4],[1],[2]),
 ([3,4],[],[1,2]),([4],[3],[1,2]),([1,4],[3],[2]),
 ([1,4],[2,3],[]),([4],[1,2,3],[]),([],[1,2,3],[4]),
 ([],[2,3],[1,4]),([2],[3],[1,4]),([1,2],[3],[4]),
 ([1,2],[],[3,4]),([2],[1],[3,4]),([],[1],[2,3,4]),
 ([],[],[1,2,3,4])]
```

Induction and Recursion over Other Data Structures

A standard way to prove properties of logical formulas is by induction on their syntactic structure. Consider e.g. the following Haskell data type for propositional formulas.

```
data Form = P Int | Conj Form Form | Disj Form Form
          | Neg Form

instance Show Form where
  show (P i) = 'P':show i
  show (Conj f1 f2) =
    "(" ++ show f1 ++ " & " ++ show f2 ++ ")"
  show (Disj f1 f2) =
    "(" ++ show f1 ++ " v " ++ show f2 ++ ")"
  show (Neg f)      = "~" ++ show f
```

It is assumed that all proposition letters are from a list $P_0, P_1, \ldots$. Then $\neg(P_1 \vee \neg P_2)$ is represented as `Neg (Disj (P 1) (Neg (P 2)))`, and shown on the screen as `~(P1 v ~P2)`, and so on.

We define the list of subformulas of a formula as follows:

```
sforms :: Form -> [Form]
sforms (P n) = [(P n)]
sforms (Conj f1 f2) =
  (Conj f1 f2): (sforms f1 ++ sforms f2)
sforms (Disj f1 f2) =
  (Disj f1 f2): (sforms f1 ++ sforms f2)
sforms (Neg f) = (Neg f):(sforms f)
```

This gives, e.g.:

```
Main> sforms (Neg (Disj (P 1) (Neg (P 2))))
[~(P1 v ~P2),(P1 v ~P2),P1,~P2,P2]
```

```
ccount :: Form -> Int
ccount (P n) = 0
ccount (Conj f1 f2) =
  1 + (ccount f1) + (ccount f2)
ccount (Disj f1 f2) =
  1 + (ccount f1) + (ccount f2)
ccount (Neg f) = 1 + (ccount f)
```

```
acount :: Form -> Int
acount (P n) = 1
acount (Conj f1 f2) = (acount f1) + (acount f2)
acount (Disj f1 f2) = (acount f1) + (acount f2)
acount (Neg f) = acount f
```

Now we can prove that the number of subformulas of a formula equals the sum of its connectives and its atoms:

Proposition 1 *For every member* `f` *of* `Form`*:*

```
length (sforms f) = (ccount f) + (acount f).
```

Proof.

Basis If `f` is an atom, then `sforms f = [f]`, so this list has length 1. Also, `ccount f = 0` and `acount f = 1`.

Induction step If `f` is a conjunction or a disjunction, we have:

- `length (sforms f) = 1 + (sforms f1) + (sforms f2)`,
- `ccount f = 1 + (ccount f1) + (ccount f2)`,
- `acount f = (acount f1) + (acount f2)`,

where `f1` and `f2` are the two conjuncts or disjuncts. By induction hypothesis:

```
length (sforms f1) = (ccount f1) + (acount f1).
length (sforms f2) = (ccount f2) + (acount f2).
```

The required equality follows immediately from this.

If `f` is a negation, we have:

- `length (sforms f) = 1 + (sforms f1)`,
- `ccount f = 1 + (ccount f1)`,
- `acount f = (acount f1)`,

and again the required equality follows immediately from this and the induction hypothesis.

□

If one proves a property of formulas by induction on the structure of the formula, then the fact is used that every formula can be mapped to a natural number that indicates its constructive complexity: 0 for the atomic formulas, the maximum of rank(Φ) and rank(Ψ) plus 1 for a conjunction $\Phi \wedge \Psi$, and so on.

Analysing Sequences by Difference Analysis

Suppose $\{a_n\}$ is a sequence of natural numbers, i.e., $f = \lambda n.a_n$ is a function in $\mathbb{N} \to \mathbb{N}$. The function f is a *polynomial function of degree* k if f can be presented in the form

$$c_k n^k + c_{k-1} n^{k-1} + \cdots + c_1 n + c_0,$$

with $c_i \in \mathbb{Q}$ and $c_k \neq 0$.

Example: the sequence

$$[1, 4, 11, 22, 37, 56, 79, 106, 137, 172, 211, 254, 301, 352, \ldots]$$

is given by the polynomial function $f = \lambda n.(2n^2 + n + 1)$. This is a function of the second degree.

Here is the Haskell check:

```
Prelude> take 15 (map (\ n -> 2*n^2 + n + 1) [0..])
[1,4,11,22,37,56,79,106,137,172,211,254,301,352,407]
```

Consider the *difference sequence* given by the function

$$d(f) = \lambda n.a_{n+1} - a_n.$$

Haskell implementation:

```
difs :: Integral a => [a] -> [a]
difs [] = []
difs [n] = []
difs (n:m:ks) = m-n : difs (m:ks)
```

```
RAC8> difs [1,4,11,22,37,56,79,106,137,172,211,254,301]
[3,7,11,15,19,23,27,31,35,39,43,47]
```

Fact: The difference function $d(f)$ of a polynomial function f is itself a polynomial function.

Example: If $f = \lambda n.(2n^2 + n + 1)$, then:

$$\begin{aligned} d(f) &= \lambda n.(2(n+1)^2 + (n+1) + 1 - (2n^2 + n + 1) \\ &= \lambda n.4n + 3. \end{aligned}$$

Here is the Haskell check:

```
RAC8> take 15 (map (\n -> 4*n + 3) [0..])
[3,7,11,15,19,23,27,31,35,39,43,47,51,55,59]
RAC8> take 15 (difs (map (\ n -> 2*n^2 + n + 1) [0..]))
[3,7,11,15,19,23,27,31,35,39,43,47,51,55,59]
```

Fact: if f is a polynomial function of degree k then $d(f)$ is a polynomial function of degree $k-1$.

For suppose $f(n)$ is given by $c_k n^k + c_{k-1} n^{k-1} + \cdots + c_1 n + c_0$. Then $d(f)(n)$ is given by

$$\begin{aligned} c_k(n+1)^k + c_{k-1}(n+1)^{k-1} + \cdots + c_1(n+1) + c_0 \\ - (c_k n^k + c_{k-1} n^{k-1} + \cdots + c_1 n + c_0). \end{aligned}$$

It is not hard to see that $f(n+1)$ has the form $c_k n^k + g(n)$, with g a polynomial of degree $k-1$. Since $f(n)$ also is of the form $c_k n^k + h(n)$, with h a polynomial of degree $k-1$, $d(f)(n)$ has the form $g(n) - h(n)$, so $d(f)$ is itself a polynomial of degree $k-1$.

Thus, if f is a polynomial function of degree k, then $d^k(f)$ will be a constant function (a polynomial function of degree 0).

Here is a concrete example of computing difference sequences until we hit at a constant sequence:

-12		**-11**		**6**		**45**		**112**	
	1		17		39		67		101
		16		22		28		34	
			6		6		6		6

213		**354**		**541**		**780**		
	141		187		239			
		46		52				
	6		6					

We find that the sequence of third differences is constant, which means that the form of the original sequence is a polynomial of degree 3.

To find the next number in the sequence, just take the sum of the last elements of the rows. This gives $6 + 52 + 239 + 780 = 1077$.

Charles Babbage (1791–1871), one of the founding fathers of computer science, used these observations in the design of his *difference engine*. We will give a Haskell version of the machine.

If the input list has a polynomial form of degree k, then after k steps of taking differences the list is reduced to a constant list:

```
RAC8> difs [-12,-11,6,45,112,213,354,541,780]
[1,17,39,67,101,141,187,239]
RAC8> difs [1,17,39,67,101,141,187,239]
[16,22,28,34,40,46,52]
RAC8> difs [16,22,28,34,40,46,52]
[6,6,6,6,6,6,6]
```

The following function keeps generating difference lists until the differences get constant:

```
difLists ::  Integral a => [[a]] -> [[a]]
difLists [] = []
difLists l@(xs:xss) =
  if constant xs then l else difLists ((difs xs):l)
  where
  constant (n:m:ms) = all (==n) (m:ms)
  constant  _       = error "lack of data/not a pf"
```

This gives the lists of all the difference lists that were generated from the initial sequence, with the constant list upfront.

```
IAR> difLists [[-12,-11,6,45,112,213,354,541,780]]
[[6,6,6,6,6,6],
 [16,22,28,34,40,46,52],
 [1,17,39,67,101,141,187,239],
 [-12,-11,6,45,112,213,354,541,780]]
```

The list of differences can be used to generate the next element of the original sequence: just add the last elements of all the difference lists to the last element of the original sequence. In our example case, to get the next element of the list

$$[-12, -11, 6, 45, 112, 213, 354, 541, 780]$$

add the list of last elements of the difference lists (including the original list): $6 + 52 + 239 + 780 = 1077$. To see that this is indeed the next element, note that $1077 - 780 = 297$, $297 - 239 = 58$, $58 - 52 = 6$, so the number 1077 'fits' the difference analysis.

The following function gets the list of last elements that we need (in our example case, the list `[6,52,239,780]`):

```
genDifferences :: Integral a => [a] -> [a]
genDifferences xs = map last (difLists [xs])
```

A new list of last elements of difference lists is computed from the current one by keeping the constant element d_1, and replacing each d_{i+1} by $d_i + d_{i+1}$.

```
nextD :: Integral a => [a] -> [a]
nextD [] = error "no data"
nextD [n] = [n]
nextD (n:m:ks) = n : nextD (n+m : ks)
```

The next element of the original sequence is given by the last element of the new list of last elements of difference lists:

```
next :: Integral a => [a] -> a
next = last . nextD . genDifferences
```

In our example case, this gives:

```
IAR> next [-12,-11,6,45,112,213,354,541,780]
1077
```

All this can now be wrapped up in a function that continues any list of polynomial form, provided that enough initial elements are given as data:

```
continue :: Integral a => [a] -> [a]
continue xs = map last (iterate nextD differences)
  where
  differences = nextD (genDifferences xs)
```

This uses the predefined `iterate` function:

```
iterate          :: (a -> a) -> a -> [a]
iterate f x       = x : iterate f (f x)
```

If a given list is generated by a polynomial, then the degree of the polynomial can be computed by difference analysis, as follows:

```
degree :: Integral a => [a] -> Int
degree xs = length (difLists [xs]) - 1
```

The difference engine is smart enough to be able to continue a list of sums of squares, or a list of sums of cubes:

```
RAC8> take 10 (continue [1,5,14,30,55])
[91,140,204,285,385,506,650,819,1015,1240]
RAC8> take 10 (continue [1,9,36,100,225,441])
[784,1296,2025,3025,4356,6084,8281,11025,14400,18496]
```

Gaussian Elimination

Difference analysis yields an algorithm for continuing any finite sequence with a polynomial form. Is it also possible to give an algorithm for finding the form? This would solve the problem of how to guess the closed forms for the functions that calculate sums of squares, sums of cubes, and so on. The answer is 'yes', and the method is Gaussian elimination.

See the lecture notes for this. We just give a demo ....

The Question about `map` and `filter` again

Consider the following problem:

Show that for every finite **and infinite** list `xs :: [a]`, every function `f :: a -> b`, every total predicate `p :: b -> Bool` the following holds:

$$\text{filter } p \text{ (map } f \text{ xs)} = \text{map } f \text{ (filter } (p.f) \text{ xs)}.$$

Now induction on the length of the list does not work anymore. How should we go about this?

We take one step back, and look at the method for defining infinite lists (and other infinite datastructures).

Corecursion

Generating streams (infinite lists) is done by means of definitions that look like recursive definitions but that lack a base case:

```
ones = 1 : ones
```

Such definitions are called **corecursive definitions**.

```
enum_1 n = n : enum_1 (n+1)
naturals = enum_1 0
```

```
enum_2 n = n : enum_2 (n+2)
odds = enum_2 1
```

iterate

From the Haskell prelude:

```
iterate           :: (a -> a) -> a -> [a]
iterate f x        = x : iterate f (f x)
```

```
ones2 = iterate id 1

theNats :: [Integer]
theNats = iterate succ 0

theOdds :: [Integer]
theOdds = iterate (\ n -> n+2) 1
```

zipWith

```
zipWith                 :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)    = z a b : zipWith z as bs
zipWith _ _      _         = []
```

In a picture:

$$[(z\ x_0\ y_0), (z\ x_1\ y_1), \ldots, (z\ x_n\ y_n), \begin{cases} [x_{n+1}, x_{n+2}, \ldots \\ [y_{n+1}, y_{n+2}, \ldots \end{cases}$$

```
theNats1 = 0 : zipWith (+) ones theNats1
```

Generating the Fibonacci numbers with `zipWith`

```
theFibs = 0 : 1 : zipWith (+) theFibs (tail theFibs)
```

```
RAC8> take 15 theFibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]
```

Eratostenes' Sieve

The definition of the sieve of Eratosthenes also uses corecursion:

```
sieve :: [Integer] -> [Integer]
sieve (0 : xs) = sieve xs
sieve (n : xs) = n : sieve (mark (xs, n-1, n-1))
  where
  mark (x : xs, 0, m) = 0 : mark (xs, m, m)
  mark (x : xs, n, m) = x : mark (xs, n-1, m)
```

Faster:

```
sieve' :: [Integer] -> [Integer]
sieve' (x:xs) = x :
   sieve' (filter (\ n -> (rem n x) /= 0) xs)

primes' :: [Integer]
primes' = sieve' [2..]
```

Proving Properties of Corecursive Programs

How does one prove things about corecursive programs? E.g., how does one prove that `sieve` and `sieve'` compute the same stream result for every stream argument? Proof by induction does not work here, for there is no base case.

Comparing the observational behaviour of infinite objects

To compare two streams xs and ys, intuitively it is enough to compare their observational behaviour. The key observation on a stream is to inspect its head. If two streams have the same head, and their tails have the same observational behaviour, then they are equal.

The two key observations on an infinite binary tree are to inspect the labels of its left and right daughters. If two infinite binary trees have the same left daughter label, the same right daughter label, and the left and right daughters have the same observational behaviour, then they are equal. And so on, for other infinite data structures.

The tool for comparing observational behaviour are bisimulations.

Bisimulations

A **bisimulation** between two sets A and B is a relation R with the following properties. If aRb then:

1. If $a \xrightarrow{o} a'$ then there is a $b' \in B$ with $b \xrightarrow{o} b'$ and $a'Rb'$.
2. If $b \xrightarrow{o} b'$ then there is an $a' \in A$ with $a \xrightarrow{o} a'$ and $a'Rb'$.

A bisimulation between A and A is called a **bisimulation on** A.

Use $\sim$ for the greatest bisimulation on a given set A.

Call two elements of A bisimilar when they are related by a bisimulation on A. Being bisimilar then coincides with being related by the greatest bisimulation:

$$a \sim b \;\Leftrightarrow\; \exists R(R \text{ is a bisimulation, and } aRb).$$

Proof by Coinduction

To show that two infinite objects x and y are equal, we show that they exhibit the same behaviour, i.e. we show that $x \sim y$. Such a proof is called a **proof by coinduction**.

The general pattern of a proof by coinduction of $x \sim y$, where $x, y :: a$, is as follows. Define a relation R on objects of some set A with $a \subset A$. Next, show that R is a bisimulation, with xRy.

Proof Recipe for a Proof by Coinduction

Showing that $x \sim y$ is done as follows:

Given: ...

To be proved: $x \sim y$

Proof:

Let R be given by ... and suppose xRy.

To be proved: R is a bisimulation.

Proof:

Suppose $x \xrightarrow{o} x'$.

To be proved: There is a y' with $y \xrightarrow{o} y'$ and $x'Ry'$.

Proof: ...

Suppose $y \xrightarrow{o} y'$.

To be proved: There is an x' with $x \xrightarrow{o} x'$ and $x'Ry'$.

Proof: ...

Thus R is a bisimulation with xRy.

Thus $x \sim y$.

Example Proof by Coinduction

$$\mathsf{map}\ f\ (\mathsf{iterate}\ f\ x) \sim \mathsf{iterate}\ f\ (f\ x).$$

Let S be the following relation on $[a]$.

$$\{(\ \mathsf{map}\ f\ (\mathsf{iterate}\ f\ x), \mathsf{iterate}\ f\ (f\ x)) \mid f :: a \to a, x :: a\}.$$

Let R be the relation $S \cup \Delta_a$ on $[a] \cup a$ (note that Δ_a is the identity on a).

Suppose:

$$(\mathsf{map}\ f\ (\mathsf{iterate}\ f\ x))\ R\ (\mathsf{iterate}\ f\ (f\ x)).$$

We show that R is a bisimulation.

$$\begin{aligned}\mathsf{map}\ f\ (\mathsf{iterate}\ f\ x) &\overset{\mathsf{iterate}}{=} \mathsf{map}\ f\ x : (\mathsf{iterate}\ f\ (f\ x)) \\ &\overset{\mathsf{map}}{=} (f\ x) : \mathsf{map}\ f\ (\mathsf{iterate}\ f\ (f\ x))\end{aligned}$$

$$\mathsf{iterate}\ f\ (f\ x) \overset{\mathsf{iterate}}{=} (f\ x) : \mathsf{iterate}\ f\ (f\ (f\ x)).$$

This shows:

$$\mathsf{map}\ f\ (\mathsf{iterate}\ f\ x) \xrightarrow{\mathsf{head}} (f\ x) \tag{1}$$

$$\mathsf{map}\ f\ (\mathsf{iterate}\ f\ x) \xrightarrow{\mathsf{tail}} \mathsf{map}\ f\ x : (\mathsf{iterate}\ f\ (f\ x)) \tag{2}$$

$$\mathsf{iterate}\ f\ (f\ x) \xrightarrow{\mathsf{head}} (f\ x) \tag{3}$$

$$\mathsf{iterate}\ f\ (f\ x) \xrightarrow{\mathsf{tail}} \mathsf{iterate}\ f\ (f\ (f\ x)) \tag{4}$$

The two observations we can perform on streams are `head` and `tail`.

Now $(f\ x)\Delta_a(f\ x)$, so $(f\ x)R(f\ x)$. Thus, the bisimilarity requirements hold for head observations.

Also,

$$(\text{map } f\ x : (\text{iterate } f\ (f\ x)))\ S\ (\text{iterate } f\ (f\ (f\ x))),$$

by definition of S, so

$$(\text{map } f\ x : (\text{iterate } f\ (f\ x)))\ R\ (\text{iterate } f\ (f\ (f\ x))),$$

by the definition of R. Hence, the bisimilarity requirements hold for tail observations.

This shows that R is a bisimimulation that connects map f (iterate $f\ x$) and iterate $f\ (f\ x)$.

Hence

$$(\text{map } f\ (\text{iterate } f\ x))\ \sim\ (\text{iterate } f\ (f\ x)).$$