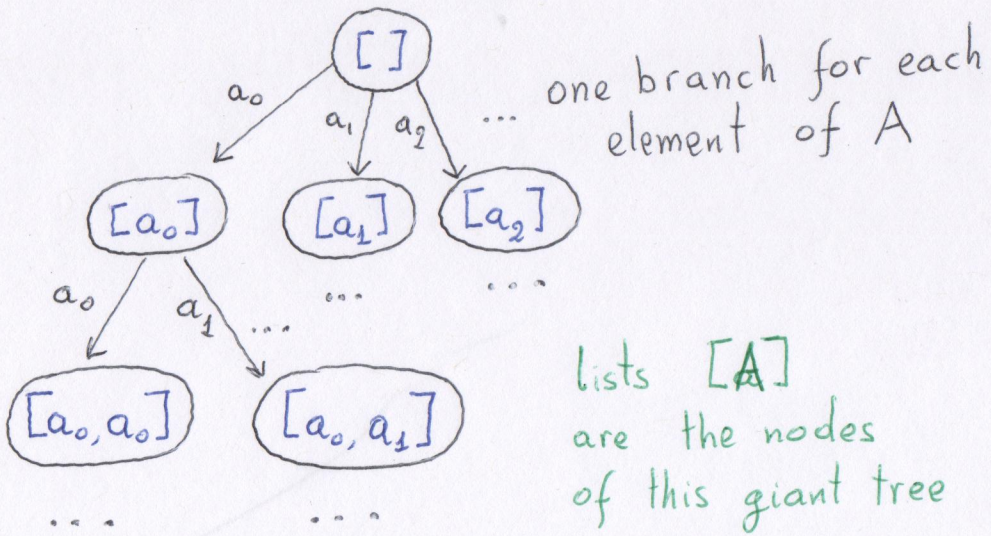
MGS 2013: Coalgebras and Infinite Data Structures

Venanzio Capretta

Lecture 3: Function Tabulation

A function on lists $f : [A] \longrightarrow B$ can be represented as a tree of its values.

Lists $[A]$ can be arranged in an infinite A-branching tree:

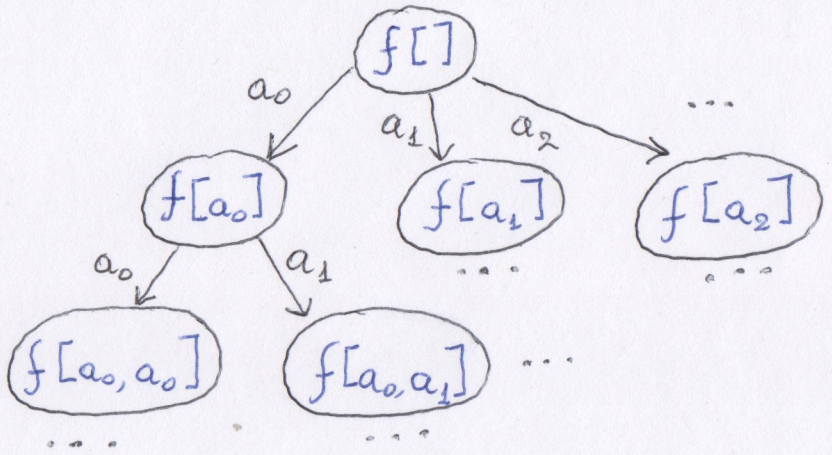


one branch for each element of A

lists $[A]$ are the nodes of this giant tree

f assigns a B value to every list
i.e. to every node

We can represent f by a tree with the same structure and B values on the nodes



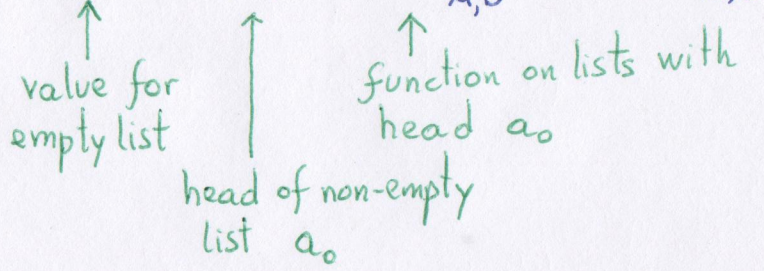
Also: every such tree with B values in the nodes defines a function

To find the value for a list use the list as a path inside the tree.
The node you reach is the result.

The set of all such trees
is a coinductive type:

```
CoInductive LFun (A,B:Set) : Set

lfun : B → (A → LFun_{A,B}) → LFun_{A,B}
```



Evaluation function: by recursion on the list

```
eval : LFun_{A,B} → [A] → B

eval (lfun b g) [] = b

eval (lfun b g) (a₀ : ℓ)
        = eval (g a₀) ℓ
```

Viceversa: Tabulation function
by corecursion on trees

```
tabulate : ([A] → B) → LFun_{A,B}

tabulate f = lfun (f [])
                  (λa. tabulate
                       (λℓ. f(a:ℓ)))
```

This function does what we illustrated:

labels every node with
the value of f on the
list corresponding to the path
to that node

Tabulation of functions f on streams

$$f : \mathbb{S}_A \longrightarrow B$$

Can we represent f as some kind of tree?

We need a constructive assumption

Brouwer's Continuity Principle:

All functions are continuous.

In the case of functions on streams

f continuous if

for every $s : \mathbb{S}_A$ there exists $k_s : \mathbb{N}$

such that

for every $s' : \mathbb{S}_A$, if the first

k_s elements of s' are the same

as those of s,

then $f\,s' = f\,s$.

We can justify the continuity principle on computational grounds.

If f is a program

when we compute $f\,s$ we get a result

in a finite number of steps.

During this computation only a finite

numer, k_s, of elements of s can

have been used.

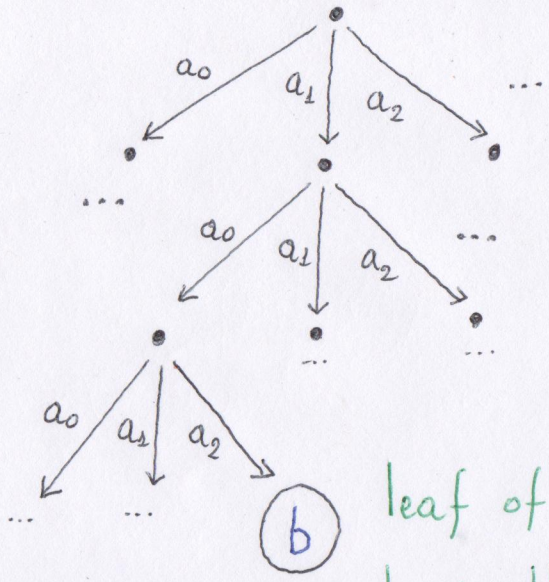
If s' coincide with s on those elements

then the computation of $f\,s'$

will be the same as that of $f\,s$.

Exploiting the Continuity Principle

We can represent f as a tree:



leaf of the tree
b is the value of fs
for every stream s
whose first three elements
are a_1, a_0, a_2

The continuity principle tells us that every path in the tree reaches a leaf in a finite number of steps.

So the tree is well-founded.

The set of such trees is an inductive type

```
Inductive SFun (A,B:Set) : Set
```

write : $B \longrightarrow SFun_{A,B}$
↑ value given in a leaf

read : $(A \longrightarrow SFun_{A,B}) \longrightarrow SFun_{A,B}$
↑ first element a_0 of the stream
↑ how to continue the computation on streams starting with a_0

Evaluation Function: by recursion on the tree:

eval : $SFun_{A,B} \longrightarrow \S_A \longrightarrow B$

eval (write b) s = b

eval (read g) s = eval ($g\ ^h s$) $^t s$

Haskell Programming : Tabulations

from U. Berger and M. Escardó
Seemingly impossible functional programs

We want to compute the universal quantification of a predicate on streams of booleans:

$$\text{allStr} : (\mathbb{S}_{\mathbb{B}} \to \mathbb{B}) \to \mathbb{B}$$

allStr f = true if (fs) = true for every stream s : $\mathbb{S}_{\mathbb{B}}$

It seems impossible to compute this function

We need to check the value of f on the infinite set of streams.

But exploiting the continuity principle we can do it in finite time.

Viceversa: Tabulation of a function on streams

- We must assume the Continuity Principle
- Assume we can check if a function is constant.

$$\text{tabulate} : (\mathbb{S}_A \to B) \to \text{SFun}_{A,B}$$

tabulate f
= if f is constant
then write (f s)
else read (λa. tabulate (λs. f(a<s)))

How can we check if f is constant ?
This is in general undecidable.

But we need only to check if f (program) is intensionally constant: it computes the result without looking at the input

If f is not always true, then we should find a counterexample on which it is false.

$$allStr : (\mathbb{S}_{\mathbb{B}} \longrightarrow \mathbb{B}) \longrightarrow \mathbb{B}$$

$$allStr\ f = f\ (counterexample\ f)$$

If f is always true, (counterexample f) will be a sequence of False, and obviously not a real counterexample.

$$counterexample : (\mathbb{S}_{\mathbb{B}} \longrightarrow \mathbb{B}) \longrightarrow \mathbb{S}_{\mathbb{B}}$$

$$counterexample\ f = \text{if } (allStr\ f_T) \text{ then } s_T \text{ else } s_F$$

where

$$f_T = \lambda s.\ f(true : s)$$
$$f_F = \lambda s.\ f(false : s)$$
$$s_T = counterexample\ f_T$$
$$s_F = counterexample\ f_F$$

counterexample and allStr are mutually recursive

f_T computes f on streams starting with true

$(allStr\ f_T)$ is true if f is true on all streams starting with true

Similarly for f_F:

$(allStr\ f_F)$ is true if f is true on all streams starting with false.

Exercise:

Prove that the Continuity Principle implies that allStr and counterexample always terminate

Can we use similar ideas to tabulate functions on streams ?

We have seen that we can define the tabulation if we can check whether f is intensionally constant.

Idea: Haskell is a partial language: it contains undefined objects.

Every type A has an element $\bot : A$ which is completely undefined.

In particular $\bot : \$_A$ is the undefined stream: we don't know any of its elements.

$f : \$_A \longrightarrow B$

Apply it to the undefined stream

$(f\ \bot)$ if you get a result then f didn't need any information about its input i.e. it is intentionally constant

If $(f\ \bot)$ is itself undefined, then it needs to read part of its input. Not intensionally constant

(It can still be constant even if it reads some input. But we don't care.)

In Haskell we can't directly test if an expression is undefined.

But we can catch the "undefined" error inside the IO monad

Stream Processors

(from Hancock/Pattinson/Ghani)

We can refine our tabulation of stream functions when the codomain is itself a type of streams

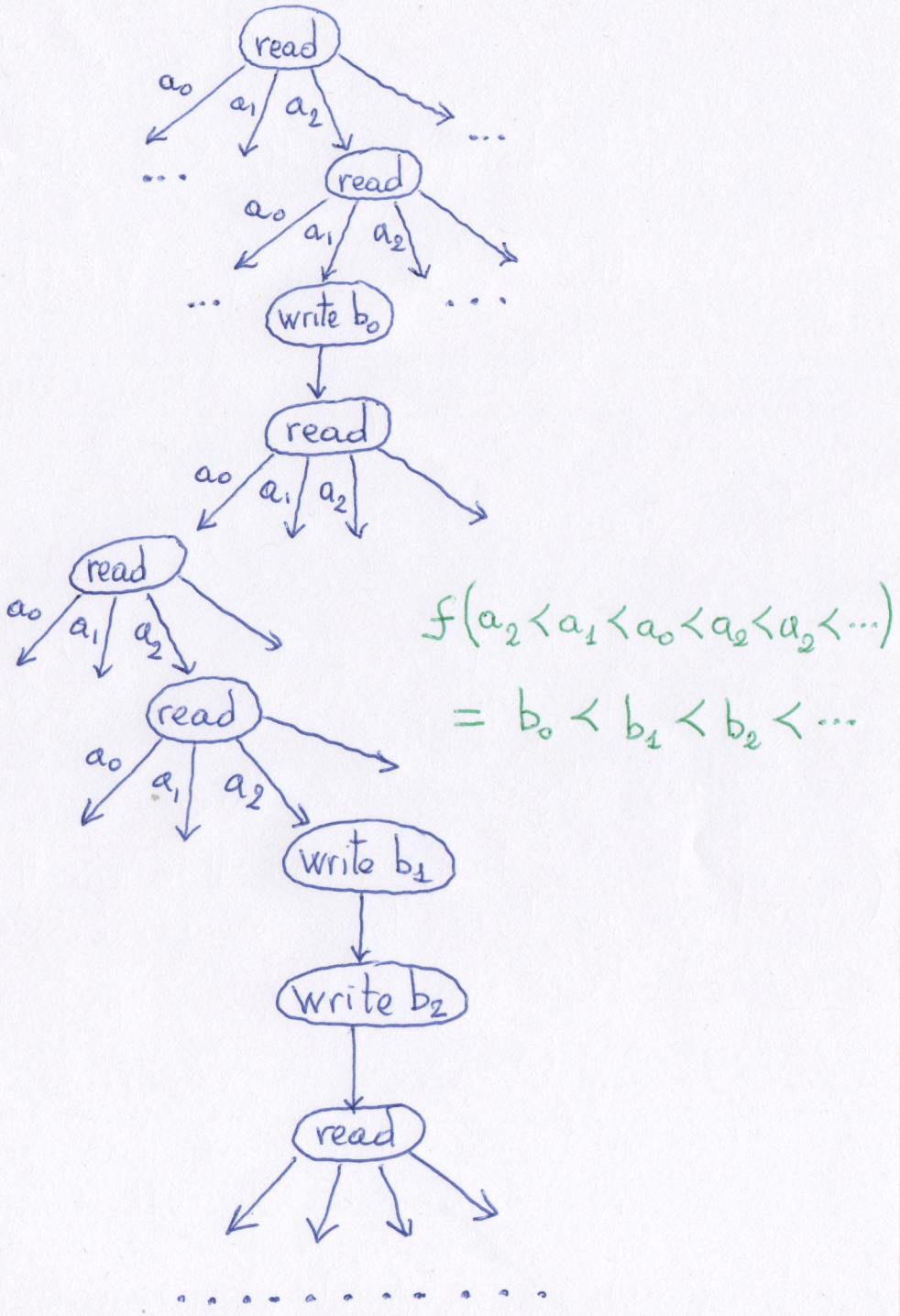
$$f : \mathbb{S}_A \longrightarrow \mathbb{S}_B$$

As before we represent f as a tree

- A-labelled "read" nodes tell us what path to follow according to the input.
- "write" nodes produce an element of the output and then continue producing the rest.

Every path is infinite (must produce infinite "write"s)

But no infinite sequence of "read"s



We need a mixed inductive/coinductive type
(see Danielsson/Altenkirch)

The "read" constructor is inductive
(no infinite sequences of reads)

The "write" constructor is coinductive
(there must be infinite writes on every path)

CoInductive $SProc_{A,B}$: Set

write : $B \longrightarrow SProc_{A,B} \longrightarrow SProc_{A,B}$

read : $(A \longrightarrow SProc_{A,B}) \longrightarrow SProc_{A,B}$

Such definitions are allowed in the system Agda

Not allowed in Coq, but we can realize them by separating the inductive and coinductive parts.

$appSP : SProc_{A,B} \longrightarrow \$_A \longrightarrow \$_B$

appSP (write b t) s

= b< (appSP t s)

↑ guarded: justified by corecursion

appSP (read g) (a<s)

= appSP (g a) s

↑ recursive call to "smaller" argument
justified by recursion

Haskell: Stream Processors

Exercise: Define a tabulation function for stream processors.
(You will need the IO monad).