MGS 2013: Coalgebras and Infinite Data Structures
Venanzio Capretta

Lecture 3: Function Tabulation

A function on lists $f: [A] \rightarrow B$ 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:

$$\text{Colinductive } \text{LFun} (A,B:\text{Set}) : \text{Set}$$

$$\text{lfun} : B \rightarrow (A \rightarrow \text{LFun}_{A,B}) \rightarrow \text{LFun}_{A,B}$$

- value for empty list
- function on lists with head \(a_0\)
- head of non-empty list \(a_0\)

Evaluation function: by recursion on the list

$$\text{eval} : \text{LFun}_{A,B} \rightarrow [A] \rightarrow B$$

$$\text{eval } (\text{lfun } b \ g) \ [] = b$$
$$\text{eval } (\text{lfun } b \ g) \ (a_0 : e) = \text{eval } (g \ a_0) \ e$$

Viceversa: Tabulation function by corecursion on trees

$$\text{tabulate} : ([A] \rightarrow B) \rightarrow \text{LFun}_{A,B}$$

$$\text{tabulate } f = \text{lfun } (f \ [] \ (\lambda a. \text{tabulate } (\lambda e. f(a:e))))$$

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 on streams

\[ f : \mathbb{S}_A \rightarrow B \]

Can we represent \( f \) as some kind of tree?

We need a \textit{constructive} assumption.

\textbf{Brouwer's Continuity Principle:}

\textit{All functions are continuous.}

In the case of functions on streams

\( f \) continuous if

for every \( s : \mathbb{S}_A \) there exists \( k_s \in \mathbb{N} \)

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 number, \( k_s \), 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:

![Diagram of a tree](image)

- Leaf of the tree
- $b$ is the value of $f$ 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 $\text{SFun} \ (A, B : \text{Set}) : \text{Set}$

- **write**: $B \rightarrow \text{SFun}_{A,B}$
  - value given in a leaf

- **read**: $(A \rightarrow \text{SFun}_{A,B}) \rightarrow \text{SFun}_{A,B}$
  - how to continue the computation
  - first element of streams starting at $a_0$ of the stream with $a_0$

Evaluation Function: by recursion on the tree:

- **eval**: $\text{SFun}_{A,B} \rightarrow S_A \rightarrow B$
- **eval (write b)** $s = b$
- **eval (read g)** $s = \text{eval (g \cdot s)} \cdot s$
Vice versa: Tabulation of a function on streams

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

\[
\text{tabulate: } (S_A \rightarrow B) \rightarrow SFun_{A,B}
\]

\[
\text{tabulate } f
\]

\[
= \text{ if } f \text{ is constant then write } (fs) \text{ else read } (\lambda a. \text{ tabulate } (\lambda s. f(a < s)))
\]

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.

---

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:

\[
\text{allStr: } (S_B \rightarrow B) \rightarrow B
\]

\[
\text{allStr } f = \text{ true if } (fs) = \text{true for every stream } s: S_B
\]

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

\[
\text{allStr : } (S_B \rightarrow B) \rightarrow B
\]

\[
\text{allStr } f = f \text{ (counterexample } f)\]

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

\[
\text{counterexample : } (S_B \rightarrow B) \rightarrow S_B
\]

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

where

\[
f_T = \lambda s. f \text{ (true : s)}
\]

\[
f_F = \lambda s. f \text{ (false : s)}
\]

\[
S_T = \text{ counterexample } f_T
\]

\[
S_F = \text{ counterexample } f_F
\]

counterexample and allStr are mutually recursive

\[
f_T \text{ computes } f \text{ on streams starting with true}
\]

\[
(\text{allStr } f_T) \text{ is true if } f \text{ is true on all streams starting with true}
\]

Similarly for $f_F$:

\[
(\text{allStr } f_F) \text{ is true if } f \text{ is true on all streams starting with false.}
\]

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 $\bot : S_A$ is the undefined stream: we don’t know any of its elements.

$f : S_A \rightarrow B$

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 intensionally 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

Haskell Programming: Tabulation of stream functions
Stream Processors
(from Hancock/Pattinson/Ghani)

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

\[ f : S_A \rightarrow S_B \]

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)

\[
\text{CoInductive } SProc_{A,B} : \text{Set}
\]

\[
\text{write : } B \rightarrow SProc_{A,B} \rightarrow SProc_{A,B}
\]

\[
\text{read : } (A \rightarrow SProc_{A,B}) \rightarrow SProc_{A,B}
\]

Such definitions are allowed in the system **Agda**

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

\[
\text{appSP : } SProc_{A,B} \rightarrow S_A \rightarrow S_B
\]

\[
\text{appSP (write b t) s} = b \langle (\text{appSP t s}) \uparrow \rangle \text{ guarded: justified by corecursion}
\]

\[
\text{appSP (read g) (a<s)} = \text{appSP (g a) s}
\]

\[
\text{recursive call to "smaller" argument justified by recursion}
\]

**Haskell**: Stream Processors

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