The Monadic Continuity Principle

Venanzio Capretta (with Paolo Capriotti)

FP lab
University of Nottingham, UK

Seminar, 26 March 2015
Institute of Cybernetics
Tallinn University of Technology, Estonia

Brouwer's Continuity Principle:

All Functions are Continuous.

L. E. J. Brouwer

Founder of Intuitionism:

Reject Platonism, Mathematical Entities are Mental Constructions.

Consider Functions on Streams of natural numbers:

$\mathbb{S}_{\mathbb{N}}$ = infinite sequences of numbers

$\sigma : \mathbb{S}_{\mathbb{N}}$, $\sigma_0, \sigma_1, \sigma_2, \ldots$ are its elements

$f : \mathbb{S}_{\mathbb{N}} \longrightarrow \mathbb{N}$ function on streams returning a number

f is continuous if it only uses a finite part of its input to compute a result.

$(f\,\sigma)$ depends only on $\sigma_0, \sigma_1, \ldots, \sigma_{n-1}$ for some n.

If σ' is another stream that coincides with σ on the first n elements,

$$\sigma'_0 = \sigma_0\ , \sigma'_1 = \sigma_1, \ldots, \sigma'_{n-1} = \sigma_{n-1},$$

then $(f\,\sigma') = (f\,\sigma)$

Notation: $\overline{\sigma}\,n = [\sigma_0, \ldots, \sigma_{n-1}]$

List of first n elements of σ

Continuity Principle:

$$\forall \sigma.\ \exists n.$$
$$\forall \sigma'.\ \overline{\sigma'}\,n = \overline{\sigma}\,n \longrightarrow f\sigma' = f\sigma$$

n is called modulus of continuity of f for σ.

Computational Justification:

If f is a computable function / program, the computation of $(f\,\sigma)$ will have a finite number of steps.
So only a finite number of elements of the input σ can have been used.

The Continuity Principle is justified constructively / computationally.

Can we assume it in Constructive Type Theory?

Martín Escardó : NO

It leads to a contradiction

Proof of Escardó's Paradox

Notation: $\sigma =_n \sigma'$ means $\overline{\sigma} n = \overline{\sigma'} n$

i.e. $\sigma_0 = \sigma'_0, \dots, \sigma'_{n-1} = \sigma_{n-1}$

Let's assume that every function

$$f : \mathbb{S}_{\mathbb{N}} \longrightarrow \mathbb{N}$$

has a modulus of continuity:

$$(\text{continuity } f) : \mathbb{S}_{\mathbb{N}} \longrightarrow \mathbb{N}$$

The Continuity Principle:

$$\forall \sigma. \forall \sigma'. \sigma' =_{(\text{continuity } \sigma)} \sigma \rightarrow (f \sigma') = (f \sigma)$$

this will lead to a contradiction

We assume that the modulus of continuity is always >0.

It's safe: we can always add 1 to it.

(this simplifies the proofs: no case analysis)

Specialize continuity to the constant zero stream:

$\mathbb{0} : \mathbb{S}_{\mathbb{N}}$ $\qquad \mathbb{0} = 0,0,0,\dots$

$$M : (\mathbb{S}_{\mathbb{N}} \longrightarrow \mathbb{N}) \longrightarrow \mathbb{N}$$

$$M\, f = \text{continuity } f\ \mathbb{0}$$

So every stream σ with the first $(M f)$ elements equal to zero, $\sigma =_{(Mf)} \mathbb{0}$, will give $(f \sigma) = (f\, \mathbb{0})$

Specialize this further: apply to the constant zero:

$$\text{const}_0 := \lambda \sigma. 0 : \mathbb{S}_{\mathbb{N}} \longrightarrow \mathbb{N}$$

$$m := M\ \text{const}_0$$

If a stream σ has the first m elements equal to 0, then $(\text{const}_0\, \sigma) = (\text{const}_0\, \mathbb{0}) = 0$

This is completely trivial!! $(\text{const}_0\, \sigma) = 0$ always!

Const_0 doesn't use any element of its input to compute its output.

So the modulus of continuity could even be zero. (we assume it's larger than zero anyway.)

We use m to construct an <u>evil function</u>.

Let $\alpha : S_{\mathbb{N}}$ be a parameter

$$\mathrm{badfun}_\alpha : S_{\mathbb{N}} \longrightarrow \mathbb{N}$$

$$(\mathrm{badfun}_\alpha\ \sigma) = \alpha_{\sigma_m}$$

Given an input σ, badfun_α computes its m^{th} element σ_m, and then returns the $\sigma_m{}^{th}$ element of α.

$$\sigma = \sigma_0, \sigma_1, \dots, \sigma_m, \dots$$

$$\alpha = \alpha_0, \alpha_1, \alpha_2, \dots, \alpha_{\sigma_m}, \dots$$

$\longrightarrow$ result

What is the modulus of continuity of badfun_α?

To compute its result badfun_α needs to evaluate σ_m.

So we may expect its modulus to be at least $m+1$.

But if α is constant, we don't need σ at all!

$$\mathrm{badf} : S_{\mathbb{N}} \longrightarrow \mathbb{N}$$

$$\mathrm{badf}\ \alpha = M\ (\mathrm{badfun}_\alpha)$$

Attention: the argument of badfun_α is σ (α is a parameter).

the argument of badf is α.

Keep this in mind when computing moduli.

If $(\mathrm{badf}\ \alpha) = n$ then:

if σ has its first n elements equal to zero, then $(\mathrm{badfun}_\alpha\ \sigma) = (\mathrm{badfun}_\alpha\ \mathbb{0})$

$$\alpha_{\sigma_m} \qquad\qquad \alpha_0$$

Observation:

$$\text{badf } \mathbb{0} = m$$

Imediate by unfolding definition and β-reduction:

$$\begin{aligned} \text{badf } \mathbb{0} &= M\ (\text{badfun}_{\mathbb{0}}) \\ &= M\ (\lambda\sigma.\ \mathbb{0}_{\sigma_m}) \\ &= M\ (\lambda\sigma. 0) = m \end{aligned}$$

this step of β-reduction is the essence of the proof
continuity depends on the reflection on computation steps.
But β-reductions are steps that happen automatically, without the possibility of reflection.

For every $\sigma, \alpha : S_{\mathbb{N}}$

$$\sigma =_{(\text{badf } \alpha)} \mathbb{0} \longrightarrow \alpha_0 = \alpha_{\sigma_m}$$

By def, $(\text{badf } \alpha) = M\ (\text{badfun}_\alpha)$
so if a stream σ coincides with $\mathbb{0}$ up to that point, then

$$\begin{array}{ccc} \text{badfun}_\alpha\ \sigma & = & \text{badfun}_\alpha\ \mathbb{0} \\ \parallel & & \parallel \\ \alpha_{\sigma_m} & & \alpha_0 \end{array}$$

To get a contradiction:

Choose σ and α s.t. $\sigma =_{(\text{badf } \alpha)} \mathbb{0}$

but $\alpha_0 = 0$, $\alpha_{\sigma_m} = 1$.

Here how to define them: suppose $(M \text{ badf}) = n+1$

$$\alpha = \underbrace{0, \ldots, \overset{\alpha_n}{\overset{\parallel}{0}}}_{n+1 \text{ entries}}, \overset{\alpha_{n+1}}{\overset{\parallel}{1}}, 1, \ldots$$

$$\sigma = \underbrace{0, \ldots, 0}_{m \text{ entries}}, \overset{\sigma_m}{\overset{\parallel}{n+1}}, n+1, \ldots$$

By def, $\sigma =_m \mathbb{0}$ i.e. (Observation) $\sigma =_{(\text{badf } \mathbb{0})} \mathbb{0}$

$\alpha =_{(M \text{ badf})} \mathbb{0}$

so (def of M) $(\text{badf } \alpha) = (\text{badf } \mathbb{0})$

in conclusion: $\sigma =_{(\text{badf } \alpha)} \mathbb{0}$

Therefore: $\alpha_0 = \alpha_{\sigma_m}$

but $\alpha_0 = 0$

$\alpha_{\sigma_m} = \alpha_{n+1} = 1$

contradiction ■

What is the source of the problem?

- Functions are <u>extensional</u>:

  the values of $f: S_{\mathbb{N}} \longrightarrow \mathbb{N}$

  depend only on the values of the elements

  of the input:

  if $\sigma_i = \sigma'_i$ for every i

  then $(f\,\sigma) = (f\,\sigma')$

- The modulus of continuity is <u>intensional</u>:

  to compute the module we must

  explicitely look at computation steps.

  Two functions may compute the

  same result, but with different

  computations.

Are streams themselves intentional or extensional?

- In the proof we exploited the intensional definition of a stream: we know $0_n = 0$ for all n, even if we don't know n.

- Brouwer: streams are extensional objects

  We don't know a rule to generate the elements.

  σ_n is known only when we require

  the input with a specific n.

We want to formalize:

Streams are given as sequential inputs

one element at a time, not as closed

λ-terms.

In Haskell, input objects are modeled

in the I/O monad.

In general:

values provided under special circumstances

(side effects, partiality, indeterminacy, ...)

are modeled inside a particular monad.

Idea: Define <u>monadic streams</u>

the head and tail of a stream

are obtained by executing a monadic

action.

Formal Definition of Streams

CoInductive $\mathbb{S}_A : Set$

$$cons : A \times \mathbb{S}_A \longrightarrow \mathbb{S}_A$$

circular definition

A stream is built from an A and a stream

This constructor must be iterated infinitely

Example of definition of stream:

$$from : \mathbb{N} \longrightarrow \mathbb{S}_{\mathbb{N}}$$

$$from\ n = cons\ n\ (from\ n+1)$$

guarded by constructor

unrestricted recursive call

$$from\ 0 = 0, 1, 2, 3, \ldots$$

Monadic Streams:

We must perform a monadic action to obtain a head and a tail

Let $M : Set \longrightarrow Set$ be a monad

CoInductive $\mathbb{S}_{M,A} : Set$

$$mcons : M\,(A \times \mathbb{S}_{M,A}) \longrightarrow \mathbb{S}_{M,A}$$

Example: reading an input stream

```
io_stream : S_{IO,A}
io_stream = mcons $
              do putStrLn "next element"
                 x <- getLine
                 return (x, io_stream)
```

(the code is a bit sloppy for exposition purposes)

Functions on monadic streams:

$$f : \forall M.\ \mathbb{S}_{M,A} \longrightarrow MB$$

functions polymorphic in the monad: we don't care how the stream is produced.

Claim: These functions must be continuous.

We need parametricity: f is defined "in the same way" on any monad.

Equivalent definition of continuity (classically)

Ghani / Hancock / Pattinson codes:

A function on $\mathbb{S}_A$ is modeled by a tree.

Leafs: output

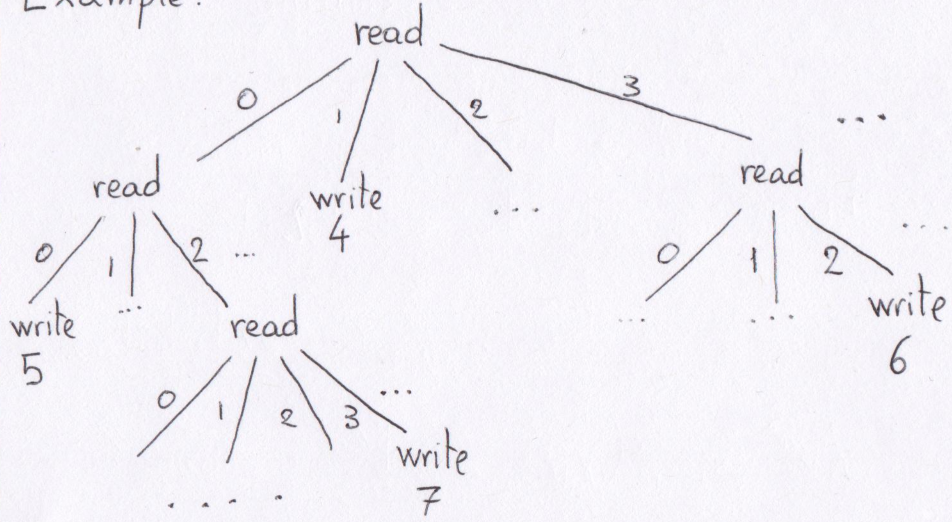
Nodes: computation steps, branching according to input element

Inductive $SF_{A,B}$: Set

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

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

Example:



This tree describes the function

$$f : \mathbb{S}_{\mathbb{N}} \longrightarrow \mathbb{N}$$

such that $f(0,0,\dots) = 5$

$f(0,2,3,\dots) = 7$

$f(1,\dots) = 4$

$f(3,2,\dots) = 6$

Since the trees are wellfounded (inductive set) the function only needs finite inputs to compute outputs: it is continuous.

The viceversa, every continuous function has a tree representation, needs a non-constructive proof.

We can <u>run</u> a tree in any monad:

$$\text{run}_M : SF_{A,B} \longrightarrow \mathbb{S}_{M,A} \longrightarrow MB$$

$$\text{run}_M (\text{write } b)\ \sigma = \text{return } b$$

inverse of constructor

$$\text{run}_M (\text{read } h)\ \sigma = \text{do } (a,\sigma') \leftarrow \text{out } \sigma$$
$$\text{run}_M (h\ a)\ \sigma'$$

Can we do the inverse?

Given $f : \forall M.\ \mathbb{S}_{M,A} \longrightarrow MB$

can we construct a tree in $SF_{A,B}$

Yes: Instantiate f with the monad SF_A itself

and $\sigma_\iota : \mathbb{S}_{SF_A, A}$

$$\sigma_\iota = \text{mcons}(\text{read } (\lambda a.\ \text{write } \langle a, \sigma_\iota \rangle))$$

Then $(f_{SF_A}\ \sigma_\iota) : SF_{A,B}$

What happens when we run this tree?

We hoped to get back f; <u>but not quite</u>!

Example:

Simple function that project the second element of the stream

$$f : \forall M.\ \mathbb{S}_{M,A} \longrightarrow MA$$

$$f_M\ \sigma = \text{do } \langle a_0, \sigma_0 \rangle \leftarrow \text{out } \sigma$$
$$\langle a_1, \sigma_1 \rangle \leftarrow \text{out } \sigma_0$$
$$\text{return } a_1$$

The corresponding tree:

$$(f_{FS_A}\ \sigma_\iota) = \text{read } (\lambda a_0.\ \text{read } (\lambda a_1.\ \text{write } a_1))$$

It is easy to check that

$$\text{run}_M\ (f_{FS_{A_0}}\ \sigma_\iota) = f_M$$

But the following counterexample (by Paolo) shows that this is not always true.

$$\mathrm{wrongf} : \forall M.\ S_{M,A} \longrightarrow MA$$

$$\begin{aligned}\mathrm{wrongf}_M\ \sigma = \mathrm{do}\ & \langle a_0, \sigma_0\rangle \leftarrow \mathrm{out}\ \sigma \\ & \langle a_1, \sigma_1\rangle \leftarrow \mathrm{out}\ \sigma \\ & \mathrm{return}\ a_1\end{aligned}$$

evaluate the same action again

This produces the same tree as the previous function:

$$(\mathrm{wrongf}_M\ \sigma_\iota) = \mathrm{read}\ (\lambda a_0.\ \mathrm{read}\ (\lambda a_1.\ \mathrm{write}\ a_1))$$

So we get the wrong result, e.g. on identity monad:

$$\mathrm{run}_{Id}\ (\mathrm{wrongf}_{Id}\ \sigma_\iota)\ \sigma = \sigma_1$$

$$\mathrm{wrongf}_{Id}\ \sigma = \sigma_0$$

Can we fix this?

Maybe we shoul only consider functions linear in the stream: they can evaluate the input stream only once.
After that, they must use the tail.

A simpler solution by
Bauer, Hofmann, Karbyshev
"On monadic parametricity of second-order functionals"

$$f : \forall M.\ MA \longrightarrow MB$$

a single monadic action that can be evaluated several times to obtain the elements of a stream

f_{SF_A} (read $\lambda a.$ write a) is the corresponding tree

Given a tree $t : SF_{A,B}$, we can run it:

$$\mathrm{run}_M : SF_{A,B} \longrightarrow MA \longrightarrow MB$$

$$\mathrm{run}_M\ (\mathrm{write}\ b)\ m = \mathrm{return}\ b$$

$$\begin{aligned}\mathrm{run}_M\ (\mathrm{read}\ h)\ m = \mathrm{do}\ & a \leftarrow m \\ & \mathrm{run}_M\ (h\ a)\ m\end{aligned}$$

This is a one-to-one correspondence (using parametricity).