A Zoo of Partial Continuity Properties in Constructive Type Theory

Internship with Yannick Forster, Cambium team, INRIA Paris

Jean Caspar

1st October 2025–28th February 2026

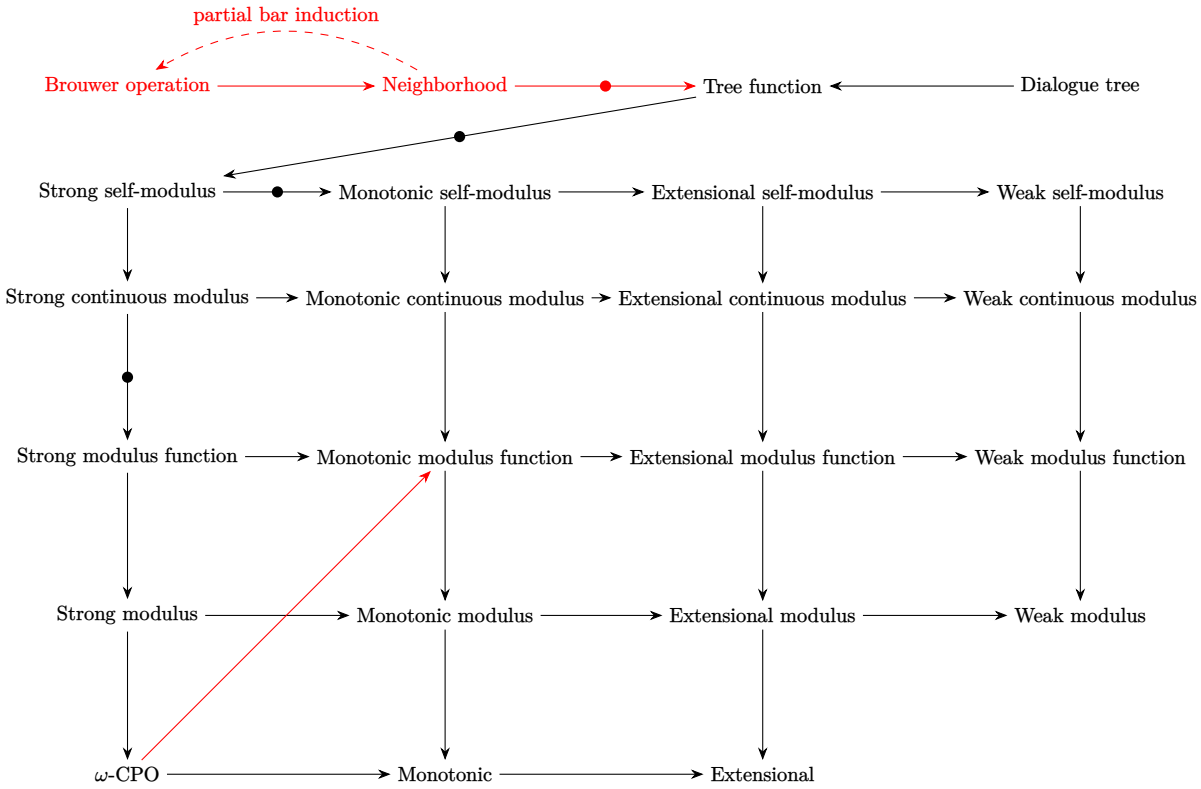


Figure 1: Summary of this work. Arrows represent implications, red elements are only valid if $Q = \mathbb{N}$, dots indicate strict implications and dashed arrows are implications which hold only if the principle above them is valid.

Contents

1 Introduction

The intuitionist school of mathematics assumes that every second-order function is continuous. However, there are several definitions of continuity in this framework, which have been classified in a uniform setting, the Rocq proof assistant, by [Bai+25]. These definitions stem from the observation that for ground types Q, A and R, computable total functions $(Q \to A) \to R$ must call their argument functions only a finite number of times. Partial recursive functions $(Q \rightharpoonup A) \rightharpoonup R$ also verify a similar property: if $F(f)$ terminates, then f must have been called a finite number of times. In my work, I define and compare different definitions of continuity for partial computable functions, then perform an analysis in the style of constructive reverse mathematics of the implication between some of them. I show that some theorem equivalent to bar induction in the total case is equivalent in this case to a custom variant of bar induction adapted to partial functions, and introduce several moduli-based definitions. Everything is formalized in the Rocq Prover[1], which features a type theory with inductive types, an infinite hierarchy of

[1]The formalization is available on github at `https://github.com/inria-cambium/internship-caspar`. The documentation is available at `https://jeancaspar.github.io/rocq/partial_functions/`. It

universes $\mathrm{Type}_i : \mathrm{Type}_{i+1} : \dots$ and an impredicative universe of propositions Prop at the bottom, which supports subsingleton elimination. I will use the notation $\mathbb{N}$, $\mathbb{B}$, $\mathbb{L}\,A$ and $\mathbb{O}\,A$ for the types of natural numbers, booleans, lists of A and options of A respectively.

First, I will present the notion of partiality I will use in section 2. Then I will present constructive reverse math in section 3, the framework of analysis I will use throughout this work to analyze some of the continuity definitions. Then, I will present the first definitions of continuity based on moduli in CPO in section 4, before presenting the different presentations of trees in section 5, followed by the different notions of tree-based continuity in section 6. Finally, in section 7, I will present an adaptation of bar induction to the partial framework and in section 8 the adaptation of Brouwer operations, together with a reverse math analysis of this definition which links it to the partial bar induction principle.

2 Partiality

The notion of partiality that interests us is the one of partial recursive functions: it must be possible to evaluate step by step a partial element. Moreover, there is a minimization operator μ, such that for each function $f : \mathbb{N} \rightharpoonup \mathbb{B}$, μf returns, if it exists, the smallest integer n such that $f(n)$ returns true and $f(k)$ returns false for every $k < n$. Furthermore, partial functions must be composable. As partiality is an effect, it seems natural to represent partial functions $A \rightharpoonup B$ as functions $A \to \mathcal{P}\,B$, with a monad $\mathcal{P}$ satisfying the aforementioned properties. This definition of partiality was given by Forster in [For21].

2.1 Abstract partiality monad

This work aims to be the most independent possible from the actual implementation of partial functions. To this end, we axiomatize the notion of "partiality monad", and the whole work is parameterized by an arbitrary partiality monad defined as follows.

Definition 1 (Partiality monad). *A partiality monad $\mathcal{P}$ is a monad* $\mathrm{Type} \to \mathrm{Type}$ *with return function* ret *and bind function* $\gg\!=$*, equipped with a relation* $\lightning : \forall(A : \mathrm{Type}), \mathcal{P}\,A \to A \to \mathrm{Prop}$*, and with functions* $\mathrm{undef} : \forall(A : \mathrm{Type}), \mathcal{P}\,A$*,* $\mu : (\mathbb{N} \rightharpoonup \mathbb{B}) \rightharpoonup \mathbb{N}$ *and* $\mathrm{seval} : \forall(A : \mathrm{Type}), \mathcal{P}\,A \to \mathbb{N} \to \mathbb{O}\,A$*, subject to the following laws, for all* $A, B : \mathrm{Type}$*,* $a, a' : A$*,* $b : B$*,* $p : \mathcal{P}\,A$*,* $f : A \rightharpoonup B$*,* $n : \mathbb{N}$*.*

- $\lightning$ *is deterministic : if* $p \lightning a$ *and* $p \lightning a'$ *then* $a = a'$*;*
- $\mathrm{ret}\,a \lightning a$*;*
- $p \gg\!= f \lightning b \iff \exists(a : A), p \lightning a \wedge f a \lightning b$*;*
- $\mathrm{undef} \not\lightning a$ *is false;*
- $\mu f \lightning n \iff (f(n) \lightning \mathrm{true}) \wedge (\forall(k < n), f(k) \lightning \mathrm{false})$*;*

is built on top of `https://github.com/amahboubi/continuity-zoo`, in a new folder named Partial.

- $p \swarrow a \iff \exists (n : \mathbb{N}), \mathrm{seval}\, p\, n = \mathrm{Some}\, a$;
- seval *is monotonic: if* $\mathrm{seval}\, p\, n = \mathrm{Some}\, a$, *then* $\mathrm{seval}\, p\, (n+1) = \mathrm{Some}\, a$.

The partial element $\mathrm{ret}\, a$ is the partial element which has a for value; $\gg\!=$ allows partial function composition; undef is the partial element with no value, μ is the minimizer and seval represents the step-evaluation mechanism.

Definition 2 (Termination). *A partial element $p : \mathcal{P}\, A$ terminates, denoted $p\!\downarrow$, if there exists $a : A$ such that $p \swarrow a$.*

Lemma 1 (Order and equivalence). *$\mathcal{P}\, A$ is a partially preordered set, for the relation $p \preceq q \iff \forall (a : A), p \swarrow a \to q \swarrow a$. Moreover, this order relation is that of an ω-CPO: monotonic sequences have a least upper bound.*

Its associated equivalence relation is $p \cong q$, defined by $\forall (a : A), p \swarrow a \iff q \swarrow a$. All monad laws hold up to $\cong$, and every function coming with the partiality monad is compatible with $\cong$, except for seval.

Most of the functions we use in this work will be extensional, that is, if their inputs are related by $\cong$, then their outputs are related, too. In fact, most of them will even be monotonic. We can also prove that the "monad laws" hold up to $\cong$ for $\mathcal{P}$.

Finally, because we are using dependent type there is a crucial function we need to define:

Lemma 2 (Termination witness). *There is a function* $\mathrm{proof\_ter} : \forall (A : \mathrm{Type})(p : \mathcal{P}\, A), \mathcal{P}(p\!\downarrow)$ *such that* $p\!\downarrow \to \mathrm{proof\_ter}\, p\!\downarrow$.

It allows one to compute a proof that p terminates for functions that need one. In general, passing dependent types inside a monad has proven to be difficult: in $p \gg\!= f$, f cannot use the fact that its input is the value of p. Therefore, I used $\mathrm{proof\_ter}\, p \gg\!= f'$ where f' uses as input $\mathrm{Hp} : p\!\downarrow$, and it can access to the value of p with $\mathrm{eval}\, \mathrm{Hp}$. eval can again be proved by a proof search that is guaranteed to terminate, and so extract its value with Markov's principle.

Proof. It is proven by searching with μ for the smallest n such that $\mathrm{seval}\, p\, n$ returns Some, and then building a proof out of it. □

2.2 Concrete implementation

In the library I used[2], the definition of partiality monad was already provided, as well as an implementation: monotonic functions $\mathbb{N} \to \mathbb{O}\, A$. However, for reasons shown below, I had to implement another partiality monad, the delay monad [Cap05], which is isomorphic to the former but is syntactically strictly positive. It is defined as the coinductive type whose definition is given in listing 1.

A term of type delay A is either an infinite term of the form $\mathrm{Delay}(\mathrm{Delay}(\ldots))$, or a finite term of the form $\mathrm{Delay}(\mathrm{Delay}(\ldots(\mathrm{Now}(a))))$, with $a : A$.

[2] https://github.com/uds-psl/coq-synthetic-computability

```
Inductive delayF (A T : Type) : Type :=
| DelayF : T -> delayF A T
| NowF : A -> delayF A T.

CoInductive delay (A : Type) := {
  delay_go : delayF A (delay A);
}.

Definition Now {A : Type} (a : A) : delay A := {| delay_go := NowF _ _ a; |}.
Definition Delay {A : Type} (d : delay A) := {| delay_go := DelayF _ _ d; |}.
```

Listing 1: The delay monad.

3 Constructive reverse mathematics

In what follows, we will consider several definitions of "continuity" for partial 2^{nd}-order functions, and try to classify them, as Baillon et al. [Bai+25] did in the total case. However, to achieve a complete classification, we must take into account the case where $\forall F, D_1(F) \to D_2(F)$ is neither provable nor unprovable for two definitions of continuity D_1 and D_2. This happens frequently, but not exclusively, in constructive settings as the type theory of Rocq.

To prove that a formula, such as $\forall F, D_1(F) \to D_2(F)$, is unprovable, but that it is consistent to assume, there are several paths: for example, one could use models, trying to find an interpretation of Rocq terms and types into a setting in which this formula holds, and another one in which it does not hold. However, this is impossible to do in Rocq, because building a model of type theory inside type theory leads to inconsistency because of the incompleteness theorem. Moreover, we would need to manipulate an encoding of the Rocq syntax inside of Rocq, which would be difficult to use.

An easier method would be to show that there are two propositions P and Q, that are both unprovable and consistent, such that $P \to (\forall F, D_1(F) \to D_2(F))$ and $(\forall F, D_1(F) \to D_2(F)) \to Q$.

But in order to analyze the formula better, we will try to find a consistent and unprovable formula P such that $P \iff (\forall F, D_1(F) \to D_2(F))$. This is called constructive reverse mathematics [Die20].

```
Definition F1 (f : nat -> nat) : nat := f 0.
Definition F2 (f : nat -> bool) : bool := if f 0 then f 1 else f 2.
Definition F3 (f : nat -> nat) : nat := f (f 0).
```

Listing 2: Three total functions that will be used as running examples.

4 Moduli and ω-CPO

The weakest notion of continuity we will see is the notion of modulus-continuous functions [TD88]. Intuitively, we know that a function $F : (Q \to A) \to R$ will only use its argument on a finite list of Q. Thus, if $F\,f$ asks only questions belonging to $l : \mathbb{L}\,Q$, and g agrees with f on l, then $F\,g$ must agree with $F\,f$. We say that l is a modulus of F at f if for every g that agrees with f on l, $F\,f$ and $F\,g$ agree. For example, in listing 2, F1 has $[0]$ as modulus at f, F2 has $[1;0]$ as modulus at f if $f(0)$ is true and $[2;0]$ if $f(0)$ is false, and F3 has $[f(0);0]$ as modulus at f.

In the partial case, it seems important to ask that $F\,f$ terminates in this definition, because else F could be asking an infinite number of questions to f.

In the partial case, we can state the definition of l being a modulus at $F(f)$ in four different ways, from the weakest to the strongest, as predicate of type $\forall(F : (Q \rightharpoonup A) \rightharpoonup R)(f : Q \rightharpoonup A), F(f)\lightning \to \mathbb{L}\,Q \to \mathrm{Prop}$[3]:

Definition 3 (Weak modulus).

$$\forall g, (\forall q \in l, \exists a, f(q) \lightning a \wedge g(q) \lightning a) \to F(g) \lightning r$$

Definition 4 (Extensional modulus).

$$\forall g, (\forall q \in l, f(q) \approx g(q)) \to F(g) \lightning r$$

Definition 5 (Monotonic modulus).

$$\forall g, (\forall q \in l, \forall a, f(q) \lightning a \to g(q) \lightning a) \to F(g) \lightning r$$

Definition 6 (Strong modulus). *f terminates on every q in l + any of the above, which are equivalent in this case.*

The first definition seems rather useless, the three others are more natural.

We can now state what it means for F to be continuous:

Definition 7 (Existential modulus). *There exists a modulus at each f such that $F(f)\lightning$.*

Definition 8 (Modulus). *There exists a function $M : (Q \rightharpoonup A) \rightharpoonup \mathbb{L}\,Q$ computing F's modulus out of f when $F(f)\lightning$.*

Definition 9 (Continuous modulus). *The function M from above itself has a modulus when it is defined.*

Definition 10 (Self-modulating modulus). *The function M from above computes its own modulus, ie. if $F\,f\lightning$ and $M\,f \lightning l$ then l is a modulus for both F and M at f.*

[3] in the definitions, r is the value of $F(f)$, defined as $\mathrm{eval}(Hf)$ with $Hf : F(f)\lightning$

In all these definitions, it is equivalent to furthermore ask $M\ f$ to be defined only when $F\ f$ is.

In each of these definitions, each "modulus" can be instantiated with different definitions of modulus. Combined with the four previous definitions of moduli, and given that the type of modulus which M and F have can differ, there are now $4+4+4\times4+4\times4=40$ definitions of continuity!

My main result is this one:

Theorem 1 (Separating some modulus-based definitions). *There exists a function that has:*

- *a monotonic modulus that is a monotonic self-modulus;*
- *a monotonic modulus that has a strong modulus;*
- *a strong modulus;*
- *but no strong modulus that is existentially monotonic modulus-continuous.*

This function is defined by taking as input $f : \mathbb{B} \rightharpoonup \mathbb{1}$ and it terminates and returns $() : \mathbb{1}$ if and only if at least one of $f(\text{true})$ and $f(\text{false})$ terminates.

We saw that $\mathcal{P}\,A$ is a (pre)ordered type that enjoys the properties of an ω-CPO. This yields another natural way to define continuity:

Definition 11 (ω-continuity). *A function $F : (Q \rightharpoonup A) \rightharpoonup R$ is ω-continuous if for every pointwise-monotonic sequence of functions $f_n : Q \rightharpoonup A$, $F(\sup_{n\in\mathbb{N}} f_n) \cong \sup_{n\in\mathbb{N}} F(f_n)$.*

This definition lies inside the hierarchy of modulus functions:

Lemma 3 (Classification of ω-continuity). *If F is existentially strong modulus-continuous, then it is ω-continuous.*

Lemma 4. *If F is ω-continuous and $Q = \mathbb{N}$, then it is monotonic modulus-continuous.*

If we leave the world of "definable Rocq functions", we can classify more definitions, but the intuition that functions can call their argument only a finite number of times fails.

Indeed, we can define the following omniscience principle, which is provable classically using choice but rejected in constructive mathematics:

Definition 12 (Computational LPO). *The computation version of LPO (limited principle of omniscience) states that there is a function that given $p : \mathcal{P}\,A$ returns* true *if and only if p terminates. Formally, it is expressed as follows, where $\{\_\} + \{\_\}$ denotes a strong sum of propositions:*

$$\forall (A : \text{Type})(p : \mathcal{P}\,A), \{p \lightning\} + \{\neg p \lightning\}$$

Lemma 5 (LPO and modulus). *LPO identifies and separates some modulus continuity definitions. Indeed, under LPO:*

- *Monotonic modulus-continuity implies strong modulus-continuity.*
- *In particular, if $Q = \mathbb{N}$, this means that monotonic modulus-continuity, ω-continuity, existentially strong modulus-continuity and strong modulus-continuity are all equivalent.*
- *There exists an extensional modulus-continuous function that is not monotonic: in particular, it is not monotonic modulus-continuous.*

Finally, the weakest "continuity" definitions of all are monotonicity, which is implied by existentially monotonic modulus-continuity, and extensionality, which is implied by existentially extensional modulus-continuity and by monotonicity. The notion of weak modulus seems a bit useless, as every list l is a weak modulus of $F\ f$ if it terminates and f is undefined on every element l.

5 Tree representations

Another way to think of Rocq-definable 2^{nd}-order functions is to see that they must ask the queries to their argument in order, as to an oracle, and that further queries may depend on previous ones. This gives these functions a tree-like representation.

5.1 Intensional and extensional trees

Trees met in programming languages or in proof assistants are often described as inductive datastructures. This definition characterizes "intensional trees". For example, an intensional A-branching tree is defined as follows:

```
Inductive tree (A : Type) :=
| Node : (A -> tree A) -> tree A
| Leaf : tree A.
```

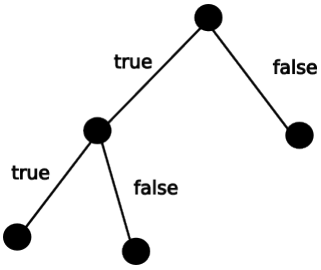
A binary tree is a $\mathbb{B}$-branching tree: each node has a child for each boolean, ie. it has two children.

However, in other branches of mathematics and computer science, trees are often represented as the set of paths from the root that belong to the tree. See for example fig. 2.

Definition 13 (Extensional tree). *An extensional A-branching tree is a predicate $T : \mathbb{L}\,A \to \text{Prop}$ that is:*

- *Suffix-closed*[4]*: $T(l' +\!\!+ l) \to T(l)$*
- *Non-empty: $T([])$ holds.*

[4]Usually, trees are asked to be prefix-closed, but my lists are reversed compared to the traditional setting.



$$T(l) := l \in \{[], [\text{true}], [\text{true}; \text{true}], [\text{false}; \text{true}], [\text{false}]\}.$$

Figure 2: A binary tree and its extensional representation.

- *Well-founded: every path "leaves" the tree:* $\forall(f : \mathbb{N} \to A)\exists(n : \mathbb{N}), \neg T([f(n-1), \ldots, f(0)])$
- *Moreover, trees are often asked to be decidable. Thus, the predicate* T *can be replaced by a boolean function* $\mathbb{L}\,A \to \mathbb{B}$.

Lemma 6. *Every intensional tree can be turned into an extensional tree.*

Proof. By induction on the tree. □

5.2 Bar induction

However, the converse implication is not provable constructively. It is equivalent to the so-called "bar induction" principle, which was coined by Brouwer [TD88] and is classically true.

To state this principle, we need the notion of bar, which can be thought of as the border of a well-founded tree (cf. fig. 3).

Definition 14 (Bar). *A predicate* $P : \mathbb{L}\,A \to \text{Prop}$ *is a bar if it meets every long enough paths:*

$$\forall(f : \mathbb{N} \to A)\exists(n : \mathbb{N}), P[f(n-1), \ldots, f(0)]$$

This means that, for example, if A is non-empty and we are given a list l, we can extend it to a list $l' +\!\!+ l$ such that the predicate holds on $l' +\!\!+ l$. Indeed, we can consider $f(n) = l[n]$ if $n < |l|$ and $f(n) = a$ with an arbitrary $a : A$ else.

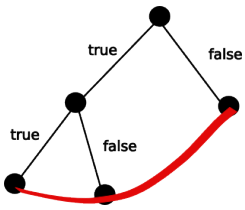


Figure 3: The border of the tree, in red, is a bar.

Moreover, as for trees, there is an inductive definition of bars.

```
Inductive inductive_bar (P : list A -> Prop) : list A -> Prop :=
| Base : forall l, P l -> inductive_bar P l
| Hered : forall l, ~ P l -> (forall (a : A), inductive_bar P (a :: l))
    -> inductive_bar P l.
```

Definition 15 (Inductive bar). *A predicate* $P : \mathbb{L}\,A \to \mathrm{Prop}$ *is an inductive bar if* inductive_bar P [] *holds.*

This definition allows one to perform proof by induction on the bar property, and especially to construct terms because this predicate can be eliminated into Type (by changing a little bit this definition to an equivalent one).

Definition 16 (Bar induction principle). *The bar induction principle is the formal statement that states that: "Every decidable bar is inductive".*

Brede and Herbelin [BH21] showed how to generalize bars and bar induction to functions $Q \to A$ and predicates $P : \mathbb{L}(Q \times A) \to \mathrm{Prop}$, which they called "generalized bar induction".

6 Tree-related definition of continuity

In this section, we will see different definitions based on the two kinds of trees of the previous section: intensional and extensional trees.

6.1 Dialogue tree

Dialogue trees [Esc13] are the most straightforward representation of 2^{nd}-order functions as trees: it is an inductive tree with nodes labelled with Q that are A-branching, and leaves labelled with R:

```
Inductive dialogue_tree (Q A R : Type) :=
| Ask : Q -> (A -> dialogue_tree Q A R) -> dialogue_tree Q A R
| Output : R -> dialogue_tree Q A R.
```

To evaluate such a tree, we follow the paths along its root. For example, to evaluate the tree d2 corresponding to F2 in fig. 4, we ask $f(0)$: if it is true, we go down the left branch, if it is false, we go down the right branch. Then, we ask $f(1)$ or $f(2)$, depending on the branch we took: this returns a boolean b, and we arrive at a leaf labelled with b, so we return b.

In code, we can represent this tree like this:

```
Definition d2 : dialogue_tree nat bool bool :=
    Ask _ _ _ 0 (fun b => if b
        then Ask _ _ _ 1 (fun b => Output _ _ _ b)
        else Ask _ _ _ 2 (fun b => Output _ _ _ b)
    ).
```

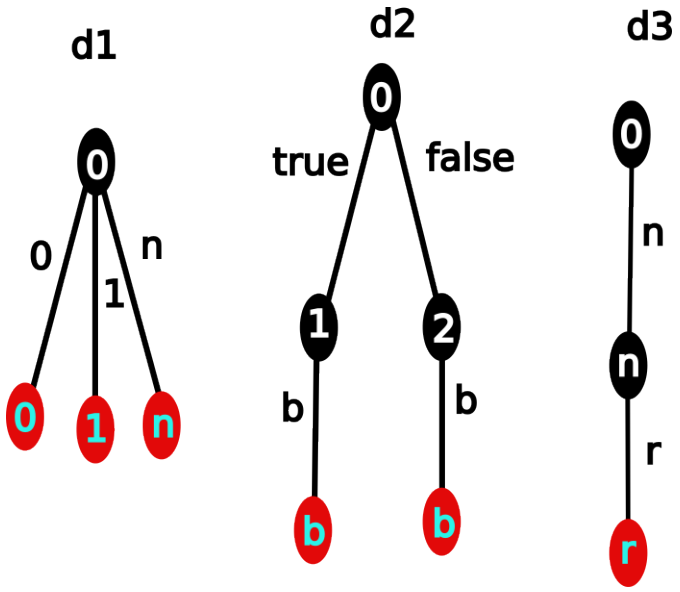


Figure 4: Dialogue trees associated to the functions in listing 2.
Black nodes correspond to queries, red ones to outputs. Label along the edges correspond to the result of the query.

Definition 17 (Dialogue tree-continuity). *A function $F : (Q \to A) \to R$ is dialogue tree-continuous if it can be presented as the evaluation function of a dialogue tree. The evaluation function of a dialogue tree is defined by induction on it.*

In the partial case, we would like for branches to be "partially defined" in the tree, so the definition should look like this:

```
Inductive dialogue_tree (Q A R : Type) :=
| Ask : Q -> (A -> partial (dialogue_tree Q A R)) -> dialogue_tree Q A R
| Output : R -> dialogue_tree Q A R.
```

However, this is not accepted by Rocq: `partial` is abstract, therefore opaque. Thus Rocq cannot verify that this definition meets the strict positivity condition that well-formed inductive types must meet. I therefore decided to implement them with a specific instantiation of the partiality monad. However, the existing implementation of partiality monad through monotonic function $\mathbb{N} \to \mathbb{O}\, A$ fails to satisfy the strict positivity criterion which validates inductive definitions. Thus, I implemented the delay monad [Cap05] (see listing 1), for which this definition works. The final definition of dialogue trees is given in listing 3.

6.2 Tree function

We can apply the idea of the extensional representation of trees to dialogue trees as well, which yields the concept of tree function [Bai+25]. A tree function is a function

```
Inductive dialogue_tree (Q A R : Type) :=
| Ask : Q -> (A -> delay (dialogue_tree Q A R)) -> dialogue_tree Q A R
| Output : R -> dialogue_tree Q A R.
```

Listing 3: Dialogue trees.

$\mathbb{L}\, A \to Q + R$ that, given a path along the dialogue tree, returns either the next question to ask, ie. the label of the node at the end of the path, or if this path encounters a leaf, it returns its label, which is the final result of the function. To make this definition "partial", I simply replaced the total function by a partial function.

Definition 18 (Tree function). *An extensional tree function is a function $\tau : \mathbb{L}\, A \rightharpoonup Q + R$ such that:*

- *for all l and l', $\tau(l' +\!\!+ l)\!\downarrow \to \tau(l)\!\downarrow$*
- *for all l, l' and r, $\tau(l) \downarrow \mathrm{inr}\, r \to \tau(l' +\!\!+ l) \downarrow \mathrm{inr}\, r$*
- *for all l, $\tau(l)\!\downarrow \implies \exists l' \exists r, \tau(l' +\!\!+ l) \downarrow \mathrm{inr}\, r$*

An extensional tree function can be used to compute a function $(Q \rightharpoonup A) \rightharpoonup R$: given as input $f : Q \rightharpoonup A$, we first evaluate $\tau([])$: if it returns $\mathrm{inr}\, r$, we return r, if it returns $\mathrm{inl}\, q$, we evaluate $f(q)$, then call its result a and start again by evaluating $\tau([a])$. The last condition in the definition of extensional tree functions, that I called strictness, is similar to a well-foundedness condition: every branch can lead to an accepting path. It is explained more in section 7. For example, the tree function corresponding to F3 is this one:

```
Definition d3 : list nat -> nat + nat :=
    fun l => match l with
    | [] => ask 0
    | [n] => ask n
    | r :: _ => ret r.
```

Definition 19 (Tree function-continuity). *A function $F : (Q \rightharpoonup A) \rightharpoonup R$ is tree function-continuous if it can be computed through a tree function.*

Lemma 7 (Dialogue tree to tree function). *Every dialogue tree-continuous function is tree function-continuous.*

I did not have the time to investigate the reverse direction. However, I expect it to follow a similar pattern to section 8.

6.3 Neighborhood function

In case $Q = \mathbb{N}$, there is a more compact representation of trees than the one we presented before: we can omit labelling question nodes and represent the question by the depth of the node in the tree. These trees have to ask questions in order: they cannot ask $f(2)$ before having asked $f(1)$ and $f(0)$. The "function version" of these trees are called neighborhood functions [Kle62].

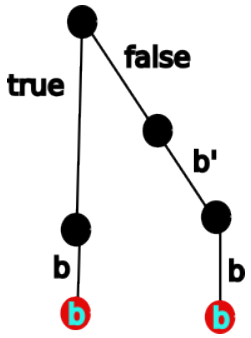


Figure 5: A neighborhood function of F2 from listing 2. An extra useless node whose answer is labelled b' had to be added.

Definition 20 (Neighborhood function). *A partial neighborhood function is a function* $\gamma : \mathbb{L}\,A \rightharpoonup \mathbb{O}\,R$ *such that:*

- *for all* l *and* l', $\gamma(l' +\!\!+ l)\lightning \to \gamma(l)\lightning$;
- *for all* l, l' *and* r, $\gamma(l)\lightning\,\mathrm{Some}\,r \to \gamma(l' +\!\!+ l)\lightning\,\mathrm{Some}\,r$;
- *for all* l, $\gamma(l)\lightning \to \exists l' \exists r, \gamma(l' +\!\!+ l)\lightning\,\mathrm{Some}\,r$.

The neighborhood function which corresponds to fig. 5 is the following:

```
Definition gamma : list bool -> option bool :=
    fun l => match l with
    | [] => None
    | [_] => None
    | [_; false] => None
    | b :: _ => Some b
    end.
```

A neighborhood γ can be used to compute a function $(Q \rightharpoonup A) \rightharpoonup R$: it takes as input $f : Q \rightharpoonup A$ and starts by asking $\gamma([])$: if it answers $\mathrm{Some}\,r$, then r is returned. Else, the function computes $f(0)$, binds its output to a_0, and asks $\gamma([a_0])$. If it outputs $\mathrm{Some}\,r$, r is returned, else the function computes $f(1)$, binds its output to a_1, and asks $\gamma([a_1, a_0])$, etc.

Definition 21 (Neighborhood continuity). *A function* $F : (Q \rightharpoonup A) \rightharpoonup R$ *is neighborhood-continuous if it can be computed through a neighborhood* γ.

Lemma 8. *Neighborhood-continuity implies tree function-continuity. Moreover, this implication is strict.*

Indeed, every neighborhood function can easily be turned into a neighborhood function which returns the question $\mathrm{inl}\,|l|$ if $\gamma(l) \not\downarrow$ None. In the total case, where neighborhood and tree functions have almost the same definitions except that they are total, we can indeed turn a tree function with $Q = \mathbb{N}$ into a neighborhood function. However, in the partial case, it is no longer possible. Take a look at the function $\lambda f.f(2)$. This function is tree-function continuous: the associated tree function simply asks the value of 2. However, it has no neighborhood: a neighborhood of this function will need to first ask the value of $f(0)$ before asking $f(2)$, which may be undefined, so the function will never return. Thus it has to be constant, which is also impossible.

We can also compare tree-continuity and modulus-based definitions of continuity:

Lemma 9 (Tree function and modulus). *Tree function-continuity implies strong self-modulating strong-continuity. The converse is provably false.*

Note that in the total case, tree function continuity, self-modulating continuity and continuous modulus-continuity are equivalent for $Q = \mathbb{N}$.

6.4 Brouwer operation

A Brouwer operation [TD88] is a special kind of neighborhood function, which is inductively defined.

Definition 22 (Brouwer operation (total case)). *A neighborhood function $\gamma : \mathbb{L}\,A \to \mathbb{O}\,R$ is a Brouwer operation at $l : \mathbb{L}\,A$ if $\exists r, \gamma(l) = \mathrm{Some}\,r$ or if $\forall (a : A), \gamma$ is a Brouwer operation at $a :: l$. A Brouwer operation is a Brouwer operation at $[]$.*

Of course, every Brouwer operation is a neighborhood function; but the converse is not provable. A constructive reverse mathematics analysis of this principle shows that the statement "every neighborhood function is a Brouwer operation" is equivalent to "every neighborhood-continuous function is Brouwer-operation continuous", but also to decidable bar induction [TD88]. During my internship, I conducted a similar analysis for the partial case.

It is not straightforward to define the notion of Brouwer operation in the partial case, so we will first design the partial version of bar induction and come back to it in section 8.

7 Partial bar induction

We want to devise an analogue of bar induction in the partial case. In the total case, using bar induction we can turn a neighborhood function for some function F into a Brouwer operation for F by using the decidable bar $P(l) := \exists r, \gamma(l) = \mathrm{Some}\,r$. However, in the partial case, $\exists r, \gamma(l) \not\downarrow \mathrm{Some}\,r$ is no longer decidable. It is decidable only if we know that $\gamma(l)$ terminates. Thus we need to introduce something to restrict the decidability requirement to l such that $\gamma(l)\!\downarrow$, which we will note $T(l)$.

Similarly, we cannot prove that P is a bar: we can only prove that:

$$\forall (f : \mathbb{N} \rightharpoonup A), F(f)\downarrow \to \exists (n : \mathbb{N}) \exists a_0, \ldots, a_{n-1}, (\forall i < n, f(i) \downarrow a_i) \wedge P([a_{n-1}, \ldots, a_0])$$

That means that our definition of a bar must also be relativised to some proposition $T(f)$, here with $T(f) := F(f)\downarrow$. We can see this T as some kind of tree: it is built on a predicate on finite paths and on a predicate on infinite (partial) paths (called ptree_path and ptree_branch in the formalization, the terminology is not standard), which enjoys some properties: if $T(f)$, $\forall (i < n), f(i) \downarrow a_i$, then $T([a_{n-1}, \ldots, a_0])$; T is closed by suffix. But it is also continuous, in the neighborhood sense: there is a neighborhood-function (γ) such that $T(f) \iff \exists n, a_0, \ldots, a_{n-1}$ such that $\forall i < n, f(i) \downarrow a_i$ and $\gamma([a_{n-1}, \ldots, a_0]) \downarrow \mathrm{Some}\,()$. This definition is not yet satisfactory, because we want to rule out the case where $T(f)$ is false but $\gamma(l) \downarrow \mathrm{None}$ for all l. In this case, an infinite number of checks using $\gamma([f(n-1), \ldots, f(0)])$ would not make us find that $T(f)$ is not true, even if the checks "progress" and f terminates on each input. Thus, we add the condition that γ is strict. However, this is not a problem, since if there is a function from $\mathbb{N}$ onto $\mathbb{L}\,A$, every neighborhood function can be turned into a strict one.

Definition 23 (Strict neighborhood function). *A neighborhood function is strict if for all l, $\gamma(l)\downarrow$ implies that $\exists r \exists l', \gamma(l' +\!\!+ l) \downarrow \mathrm{Some}\, r$.*

This can be seen as some kind of well-foundedness condition.

Definition 24 (Neighborhood-continuous semi-computable tree). *A neighborhood-continuous semi-computable tree[5] T is given by a strict neighborhood-continuous function F from which we derive two predicates $T(f)$ and $T(l)$ on partial infinite paths and on finite paths, such that $T(f) \iff F(f)\downarrow$ and $T(l) \iff \gamma(l)\downarrow$.*

Now we can define bars, and the bar induction can be stated as well:

Definition 25 (Partial bar). *Given T a neighborhood-continuous semi-computable tree, a predicate $P : \mathbb{L}\,A \to \mathrm{Prop}$ is a T-bar if for all $f : \mathbb{N} \rightharpoonup A$, $T(f)$ implies that there is some $n : \mathbb{N}$ and $a_{n-1}, \ldots, a_0$ such that for all $i < n$, $f(i) \downarrow a_i$ and $P([a_{n-1}, \ldots, a_0])$ holds.*

Definition 26 (Partial inductive bar). *A predicate $P : \mathbb{L}\,A \to \mathrm{Prop}$ is an inductive T-bar at some $l : \mathbb{L}\,A$ for $Hl : T(l)$ if $P(l)$ holds or that $\neg P(l)$ and for all $a : A$ and $Hal : T(a :: l)$, P is an inductive T-bar at $a :: l$ and Hal. P is an inductive T-bar if it is an inductive T-bar at $[]$ for all proofs of $T([])$. In particular, it is an inductive T-bar if $T([])$ is false, that is, if γ does not terminate.*

Definition 27 (Partial bar induction principle). *Given a type A, the partial bar induction principle states that for all neighborhood-continuous semi-computable tree T and $P : \mathbb{L}\,A \to \mathrm{Prop}$,*

- *if P is a T-bar;*

[5] sigma1_tree in the formalization

- *and P is T-(computably) decidable: for all l, $T(l) \to \{P(l)\} + \{\neg P(l)\}$;*[6]
- *and P is monotonic: for all l and l', $P(l) \to P(l' +\!\!+ l)$;*[7]

Then P is a T-inductive bar.

This principle is unprovable, because it implies the usual bar induction principle. Moreover, the axiom of choice seems to imply this principle, which would make it consistent. However, I have no mechanized proof of this fact, and only a sketch of a pen-and-paper proof.

8 Partial Brouwer operation

Partial Brouwer operations can be defined akin to the definition of partial inductive bar:

Definition 28 (Brouwer operation). *A neighborhood function γ is a T-Brouwer operation at $l : \mathbb{L}\,A$ and $Hl : T(l)$ if $\exists r, \gamma(l) \lightning$ Some r or if $\gamma(l) \lightning$ None and for all $a : A$ and $Hal : T(a :: l)$, γ is a T-Brouwer operation at $a :: l$ and Hal.*

Definition 29 (Brouwer operation-continuity). *A function $F : (\mathbb{N} \rightharpoonup A) \rightharpoonup R$ is Brouwer operation-continuous if there is a strict neighborhood function γ for F such that if $H : \gamma([]) \lightning$ then γ is a T-Brouwer operation at $[]$ and H for T the tree associated to F and γ.*

Of course, every Brouwer operation is a neighborhood function. We can now state the main theorem of this work:

Theorem 2. *For any type A, if there is a function from $\mathbb{N}$ onto $\mathbb{L}\,A$, the following are equivalent:*

- *The partial bar induction principle at type A holds (definition 27);*
- *Every neighborhood function γ of a function $F : (\mathbb{N} \rightharpoonup A) \rightharpoonup R$ is a Brouwer operation for F;*
- *Every neighborhood-continuous function is Brouwer operation-continuous.*

In particular, this is true for $A = \mathbb{N}$.

This analysis of the notion of partial Brouwer operations and neighborhood functions in the style of reverse mathematics concludes the main part of my work.

The definition of partial Brouwer operations is not straightforward: they are defined as an indexed inductive proposition, with brouwer_op $T\,\gamma$ having type $\forall (l : \mathbb{L}\,A), T(l) \to$ Prop. It can be easier to see them as "partial predicates", ie. as having type brouwer_op $T\,\gamma : \mathbb{L}\,A \to \mathcal{P}$ Prop, such that $T(l)$ implies that it evaluates to an actual predicate: $T(l) \to$ brouwer_op $T\,\gamma\,l \lightning$. The same goes for partial bar induction. However, defining Brouwer operations directly that way would not be accepted by Rocq (and perhaps more difficult to use).

[6] $\{A\} + \{B\}$ is the strong disjunction of A and B: there is a term which decides whether A or B holds.
[7] in the total case, the bar induction principle for monotonic bars implies the bar induction principle for decidable bars, but I did not manage to eliminate this hypothesis here.

9 Conclusion

In this work, I adapted existing definitions of continuity for second-order total functions to second-order partial recursive functions. I developed 40 definitions of continuity linked to moduli, analyzed the implications between some of them, and separated some others. I also delved into the different representations of trees in constructive mathematics and the definitions of continuity that use them. In particular, I adapted the definitions of tree functions, neighborhood functions, dialogue trees and Brouwer operations to the partial case, which lead to problems in the Rocq formalization for dialogue trees because of the strict positivity requirement. I finally devised a bar induction principle for partial bars and showed it equivalent to the fact that neighborhood-continuity implies Brouwer operation-continuity, concluding a constructive reverse math analysis of the relative strength of these two statements.

Finally, I wanted to share some thoughts about how to continue this work. Firstly, the definition of partial bar induction can obviously be extended to generalized bar induction, but the notion of tree would probably need to be changed to "tree function-continuous semi-computable tree", so this partial bar induction principle would probably be stronger than the neighborhood-continuous version. Secondly, the goal of this internship was to see what happens if we switch from total function to partial functions, which represent "pure functions" in a programming language that may loop. But what happens in a programming language with effects? This could be investigated further, but most of the definitions seem generic enough to handle effects. In order to make the monad more generic, we can see that Prop is an algebra (up to equivalence) for the monad $\mathcal{P}$, with $\mathcal{P}\,\mathrm{Prop} \to \mathrm{Prop}$ defined by $h \coloneqq \lambda X.\exists(R : \mathrm{Prop}), X \lightning R \wedge R$. Then $p \lightning a$ can simply be defined as $h(p \gg\!= (\lambda a'.\,\mathrm{ret}(a = a')))$. Maybe asking for Prop to be an algebra could be a useful generalization of the $\lightning$ predicate, but this remains to be investigated.

I am grateful to Yannick Forster for his guidance throughout this internship, for his advice on presenting mathematical work clearly, and for the freedom he gave me to explore. This internship allowed me to familiarize myself with formalization in Rocq and with constructive reverse mathematics. I also thank the Cambium team at INRIA Paris for their warm welcome.

References

[Kle62] S. C. Kleene. "S. C. Kleene. Countable functionals. Constructivity in mathematics, Proceedings of the Colloquium held at Amsterdam, 1957, Studies in logic and the foundations of mathematics, North-Holland Publishing Company, Amsterdam1959, pp. 81–100. - S. C. Kleene. Recursive functionals of higher finite types. Summaries of talks presented at the Summer Institute for Symbolic Logic, Cornell University, 1957, 2nd edn., Communications Research Division, Institute for Defense Analyses, Princeton1960, pp. 148–154." In: *The Journal of Symbolic Logic* 27.3 (Sept. 1962), pp. 359–360. ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2964658.

[TD88] Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics*. Vol. 1. Studies in Logic and the Foundations of Mathematics 121. Jan. 1, 1988. ISBN: 978-0-444-70266-1. URL: https://shop.elsevier.com/books/constructivism-in-mathematics-vol-1/troelstra/978-0-444-70266-1.

[Cap05] Venanzio Capretta. "General Recursion via Coinductive Types". In: *Logical Methods in Computer Science* Volume 1, Issue 2 (July 13, 2005). Publisher: Episciences.org. ISSN: 1860-5974. DOI: 10.2168/LMCS-1(2:1)2005.

[Esc13] Martín Escardó. "Continuity of Gödel's System T Definable Functionals via Effectful Forcing". In: *Electron. Notes Theor. Comput. Sci.* 298 (Nov. 1, 2013), pp. 119–141. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2013.09.010.

[Die20] Hannes Diener. *Constructive Reverse Mathematics*. Apr. 4, 2020. DOI: 10.48550/arXiv.1804.05495. arXiv: 1804.05495[math].

[BH21] Nuria Brede and Hugo Herbelin. "On the logical structure of choice and bar induction principles". In: *Proceedings of the Thirty sixth Annual IEEE Symposium on Logic in Computer Science (LICS 2021)*. LICS 2021 - 36th Annual Symposium on Logic in Computer Science. Rome, Italy: IEEE Computer Society Press, June 29, 2021, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470523.

[For21] Yannick Forster. "Computability in Constructive Type Theory". PhD thesis. Saarland University, 2021. URL: https://ps.uni-saarland.de/~forster/thesis/phd-thesis-yforster-screen.pdf.

[Bai+25] Martin Baillon et al. "A Zoo of Continuity Properties in Constructive Type Theory". In: *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025)*. Ed. by Maribel Fernández. Vol. 337. Leibniz International Proceedings in Informatics (LIPIcs). ISSN: 1868-8969. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 9:1–9:20. ISBN: 978-3-95977-374-4. DOI: 10.4230/LIPIcs.FSCD.2025.9.