Inductive Continuity via Brouwer Trees

Liron Cohen [1] Bruno da Rocha Paiva [2] Vincent Rahli [2]
Ayberk Tosun [2]

[1]Ben-Gurion University, Beer-Sheva, Israel

[2]University of Birmingham, UK

29 August 2023
MFCS 2023
Bordeaux, France

The Continuity Principle (1)

One cannot define a discontinuous function on the reals by computational means.

The Continuity Principle (1)

One cannot define a discontinuous function on the reals by computational means.

The computational content of such a function would involve transcending *the infinity of time*.

The Continuity Principle (1)

One cannot define a discontinuous function on the reals by computational means.

The computational content of such a function would involve transcending *the infinity of time*.

The continuity principle is the embodiment of this fact in foundations of constructive mathematics.

The Continuity Principle (2)

Define the Baire space: $\mathfrak{B} :\equiv \mathbb{N} \to \mathbb{N}$.

The Continuity Principle (2)

Define the Baire space: $\mathfrak{B} :\equiv \mathbb{N} \to \mathbb{N}$.

Consider a function $F : \mathfrak{B} \to \mathbb{N}$.

The Continuity Principle (2)

Define the Baire space: $\mathfrak{B} :\equiv \mathbb{N} \to \mathbb{N}$.

Consider a function $F : \mathfrak{B} \to \mathbb{N}$.

Conceptually, what we mean by the "continuity of F" is:

> *any result $F(\alpha)$ computed by F is determined by a finite amount of information obtained from the input α.*

Our contribution

We develop the first *internalisation* of a certain strong form of the continuity principle inside the type theory $\mathsf{TT}^{\square}_{\mathcal{C}}$ [CR22; CR23].

Our contribution

We develop the first *internalisation* of a certain strong form of the continuity principle inside the type theory $\mathrm{TT}_{\mathcal{C}}^{\square}$ [CR22; CR23].

More specifically,

- we construct a program in $\mathrm{TT}_{\mathcal{C}}^{\square}$ that realises the inductive continuity principle,
- that uses references to compute Brouwer trees.

Forms of the continuity principle

Different **forms of the continuity principle** capture continuity to varying levels of strictness.

Forms of the continuity principle

Different **forms of the continuity principle** capture continuity to varying levels of strictness.

Define $\mathfrak{B} :\equiv \mathbb{N} \to \mathbb{N}$; $\mathfrak{C} :\equiv \mathbb{N} \to \mathsf{Bool}$.

Different **forms of the continuity principle** capture continuity to varying levels of strictness.

Define $\mathfrak{B} :\equiv \mathbb{N} \to \mathbb{N}$; $\mathfrak{C} :\equiv \mathbb{N} \to$ Bool.

Continuity Principle (**Cont**):

- $\forall F : \mathfrak{B} \to \mathbb{N}. \forall \alpha : \mathfrak{B}. \exists n : \mathbb{N}. \forall \beta : \mathfrak{B}. \alpha =_n \beta \to F(\alpha) = F(\beta)$.

Forms of the continuity principle

Different **forms of the continuity principle** capture continuity to varying levels of strictness.

Define $\mathfrak{B} :\equiv \mathbb{N} \to \mathbb{N}$; $\mathfrak{C} :\equiv \mathbb{N} \to \text{Bool}$.

Continuity Principle (**Cont**):

- $\forall F : \mathfrak{B} \to \mathbb{N}. \forall \alpha : \mathfrak{B}. \exists n : \mathbb{N}. \forall \beta : \mathfrak{B}. \alpha =_n \beta \to F(\alpha) = F(\beta)$.

Uniform Continuity Principle (**UCP**):

- $\forall F : \mathfrak{C} \to \mathbb{N}. \exists n : \mathbb{N}. \forall \alpha, \beta : \mathfrak{C}. \alpha =_n \beta \to F(\alpha) = F(\beta)$.

Different **forms of the continuity principle** capture continuity to varying levels of strictness.

Define $\mathfrak{B} :\equiv \mathbb{N} \to \mathbb{N}$; $\mathfrak{C} :\equiv \mathbb{N} \to$ Bool.

Continuity Principle (**Cont**):

- $\forall F : \mathfrak{B} \to \mathbb{N}.\, \forall \alpha : \mathfrak{B}.\, \exists n : \mathbb{N}.\, \forall \beta : \mathfrak{B}.\, \alpha =_n \beta \to F(\alpha) = F(\beta)$.

Uniform Continuity Principle (**UCP**):

- $\forall F : \mathfrak{C} \to \mathbb{N}.\, \exists n : \mathbb{N}.\, \forall \alpha, \beta : \mathfrak{C}.\, \alpha =_n \beta \to F(\alpha) = F(\beta)$.

Inductive Continuity Principle (**ICP**):

- For any $F : \mathfrak{B} \to \mathbb{N}$, and any $\alpha : \mathfrak{B}$, there is a Brouwer tree whose path at α encodes the computation $F(\alpha)$.

Brouwer trees

Our construction uses Escardó's technique [Esc13] of capturing continuity information using dialogue trees.

Brouwer trees

Our construction uses Escardó's technique [Esc13] of capturing continuity information using dialogue trees.

- Except, instead of dialogue trees we use Brouwer trees.

Brouwer trees

Our construction uses Escardó's technique [Esc13] of capturing continuity information using dialogue trees.

- Except, instead of dialogue trees we use Brouwer trees.

Consider the computation $F :\equiv \lambda\alpha.\, \alpha(\underline{2})$.

Brouwer trees

Our construction uses Escardó's technique [Esc13] of capturing continuity information using dialogue trees.

- Except, instead of dialogue trees we use Brouwer trees.

Consider the computation $F :\equiv \lambda\alpha.\, \alpha(\underline{2})$.

- For input $\{0, 1, 2, \ldots\}$, it gives 2 (marked green).

Brouwer trees

Our construction uses Escardó's technique [Esc13] of capturing continuity information using dialogue trees.

- Except, instead of dialogue trees we use Brouwer trees.

Consider the computation $F :\equiv \lambda\alpha.\ \alpha(\underline{2})$.

- For input $\{0, 1, 2, \ldots\}$, it gives 2 (marked green).
- For input $\{0, 0, 0, \ldots\}$, it gives 0 (marked red).

Brouwer trees

Our construction uses Escardó's technique [Esc13] of capturing continuity information using dialogue trees.

- Except, instead of dialogue trees we use Brouwer trees.

Consider the computation $F :\equiv \lambda\alpha.\, \alpha(\underline{2})$.

- For input $\{0, 1, 2, \ldots\}$, it gives 2 (marked green).
- For input $\{0, 0, 0, \ldots\}$, it gives 0 (marked red).

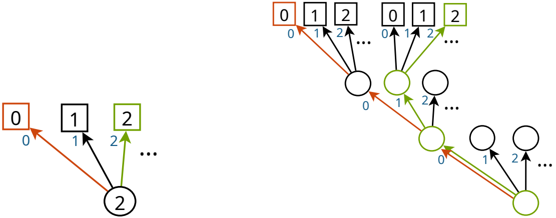

Figure: Dialogue and Brouwer tree encodings of the computation F.

The Inductive Continuity Principle

The Inductive Continuity Principle (**ICP**) says

> *For any function $F : \mathfrak{B} \to \mathbb{N}$, there is a Brouwer tree t such that for each $\alpha : \mathfrak{B}$, the path of t along α encodes the computation $F(\alpha)$.*

[1] as far as the authors are aware.

The Inductive Continuity Principle

The Inductive Continuity Principle (**ICP**) says

> *For any function $F : \mathfrak{B} \to \mathbb{N}$, there is a Brouwer tree t such that for each $\alpha : \mathfrak{B}$, the path of t along α encodes the computation $F(\alpha)$.*

- Goes back to Brouwer in intuitionistic mathematics and Kleene in classical computability theory.

[1] as far as the authors are aware.

The Inductive Continuity Principle

The Inductive Continuity Principle (**ICP**) says

> *For any function $F : \mathfrak{B} \to \mathbb{N}$, there is a Brouwer tree t such that for each $\alpha : \mathfrak{B}$, the path of t along α encodes the computation $F(\alpha)$.*

- Goes back to Brouwer in intuitionistic mathematics and Kleene in classical computability theory.
- First explicitly studied[1] by Ghani, Hancock, and Pattinson [GHP06].

[1] as far as the authors are aware.

The Inductive Continuity Principle

The Inductive Continuity Principle (**ICP**) says

> *For any function $F : \mathfrak{B} \to \mathbb{N}$, there is a Brouwer tree t such that for each $\alpha : \mathfrak{B}$, the path of t along α encodes the computation $F(\alpha)$.*

- Goes back to Brouwer in intuitionistic mathematics and Kleene in classical computability theory.
- First explicitly studied[1] by Ghani, Hancock, and Pattinson [GHP06].
- Implies both **Cont** and **UCP**.

[1] as far as the authors are aware.

Previous work

- Longley [Lon99] pioneered the idea of using effects to compute moduli of continuity.
- Coquand and Jaber [CJ12] proved that MLTT-definable functions on the Cantor space are uniformly continuous using forcing.
- Rahli and Bickford [RB18] applied Longley's method to (computational) type theory.
- Ghani, Hancock, and Pattinson [GHP06] started the study of **ICP**.
- Escardó [Esc13] used a **dialogue tree** translation for computing moduli of continuity of System T-definable functions.
- Baillon, Mahboubi, and Pédrot [BMP22] externally validated a continuity principle for a simple **intensional type theory** with restricted dependent elimination.

The computational system $\mathsf{TT}^{\square}_{\mathcal{C}}$ (1)

To internalise **ICP**, we work in the system $\mathsf{TT}^{\square}_{\mathcal{C}}$ [CR22; CR23]:

An **effectful**, extensional type theory.

The computational system $\mathrm{TT}_{\mathcal{C}}^{\square}$ (1)

To internalise **ICP**, we work in the system $\mathrm{TT}_{\mathcal{C}}^{\square}$ [CR22; CR23]:

An **effectful**, extensional type theory.

- $\mathrm{TT}_{\mathcal{C}}^{\square}$ is more general than we need here.

The computational system $\mathrm{TT}_{\mathcal{C}}^{\Box}$ (1)

To internalise **ICP**, we work in the system $\mathrm{TT}_{\mathcal{C}}^{\Box}$ [CR22; CR23]:

An **effectful**, extensional type theory.

- $\mathrm{TT}_{\mathcal{C}}^{\Box}$ is more general than we need here.
- For the purposes of our work: it is a computational type theory equipped with mutable references.

The computational system $\mathsf{TT}_{\mathcal{C}}^{\square}$ (2)

$$
\begin{array}{llllllll}
v \in \mathsf{Val} & ::= & vt & \text{(type)} & \mid & \lambda x.t & \text{(lambda)} \\
 & \mid & \underline{n} & \text{(number)} & \mid & \mathtt{inl}(t) & \text{(left inj.)} \\
 & \mid & \langle t_1, t_2 \rangle & \text{(pair)} & \mid & \mathtt{inr}(t) & \text{(right inj.)} \\
 & \mid & \star & \text{(constant)} & \mid & \delta & \text{(ref. name)}
\end{array}
$$

The computational system $\mathrm{TT}_{\mathcal{C}}^{\square}$ (2)

$$
\begin{array}{llllll}
v \in \mathsf{Val} & ::= & vt & \text{(type)} & \mid \; \lambda x.t & \text{(lambda)} \\
 & \mid & \underline{n} & \text{(number)} & \mid \; \mathtt{inl}(t) & \text{(left inj.)} \\
 & \mid & \langle t_1, t_2 \rangle & \text{(pair)} & \mid \; \mathtt{inr}(t) & \text{(right inj.)} \\
 & \mid & \star & \text{(constant)} & \mid \; \delta & \text{(ref. name)}
\end{array}
$$

$$
\begin{array}{llllll}
v \in \mathsf{Type} & ::= & \Pi_{x:t_1} t_2 & \text{(product)} & \mid \; \mathbb{U}_i & \text{(universe)} \\
 & \mid & \Sigma_{x:t_1} t_2 & \text{(sum)} & \mid \; t_1 = t_2 \in t & \text{(equality)} \\
 & \mid & \{x : t_1 \mid t_2\} & \text{(set)} & \mid \; \|t\| & \text{(truncation)} \\
 & \mid & \mathtt{Nat} & \text{(naturals)} & \mid \; t_1 \cap t_2 & \text{(intersection)} \\
 & \mid & t_1 + t_2 & \text{(disj. union)} & \mid \; \mathtt{pure} & \text{(pure)}
\end{array}
$$

The computational system $TT_{\mathcal{C}}^{\square}$ (2)

$v \in$ Val	$::=$	vt	(type)	$\mid$	$\lambda x.t$	(lambda)
	$\mid$	$\underline{n}$	(number)	$\mid$	$\texttt{inl}(t)$	(left inj.)
	$\mid$	$\langle t_1, t_2 \rangle$	(pair)	$\mid$	$\texttt{inr}(t)$	(right inj.)
	$\mid$	$\star$	(constant)	$\mid$	δ	(ref. name)
$v \in$ Type	$::=$	$\Pi_{x:t_1} t_2$	(product)	$\mid$	$\mathbb{U}_i$	(universe)
	$\mid$	$\Sigma_{x:t_1} t_2$	(sum)	$\mid$	$t_1 = t_2 \in t$	(equality)
	$\mid$	$\{x : t_1 \mid t_2\}$	(set)	$\mid$	$\lVert t \rVert$	(truncation)
	$\mid$	$\texttt{Nat}$	(naturals)	$\mid$	$t_1 \cap t_2$	(intersection)
	$\mid$	$t_1 + t_2$	(disj. union)	$\mid$	$\texttt{pure}$	(pure)
$t \in$ Term	$::=$	x	(variable)	$\mid$	$!t$	(read)
	$\mid$	v	(value)	$\mid$	$\nu x.t$	(fresh)
	$\mid$	$t_1 t_2$	(application)	$\mid$	$t_1 := t_2$	(write)
	$\mid$	$\texttt{fix}(t)$	(fixed point)	$\mid$	$t_1 \cap t_2$	(intersection)
	$\mid$	$t_1 <_? t_2$	(less than)	$\mid$	$t_1 =_? t_2$	(equality)
	$\mid$	let $x, y = t_1$ in t_2	(pair destr.)	$\mid$	let $x = t_1$ in t_2	(cbv)
	$\mid$	$t_1 + t_2$	(addition)			

The computational system $TT_{\mathcal{C}}^{\Box}$ (2)

$v \in$ Val	$::=$	vt	(type)	$\mid$	$\lambda x.t$	(lambda)
	$\mid$	$\underline{n}$	(number)	$\mid$	$\mathtt{inl}(t)$	(left inj.)
	$\mid$	$\langle t_1, t_2 \rangle$	(pair)	$\mid$	$\mathtt{inr}(t)$	(right inj.)
	$\mid$	$\star$	(constant)	$\mid$	δ	(ref. name)
$v \in$ Type	$::=$	$\Pi_{x:t_1} t_2$	(product)	$\mid$	$\mathbb{U}_i$	(universe)
	$\mid$	$\Sigma_{x:t_1} t_2$	(sum)	$\mid$	$t_1 = t_2 \in t$	(equality)
	$\mid$	$\{x : t_1 \mid t_2\}$	(set)	$\mid$	$\lVert t \rVert$	(truncation)
	$\mid$	$\mathtt{Nat}$	(naturals)	$\mid$	$t_1 \cap t_2$	(intersection)
	$\mid$	$t_1 + t_2$	(disj. union)	$\mid$	$\mathtt{pure}$	(pure)
$t \in$ Term	$::=$	x	(variable)	$\mid$	$!t$	(read)
	$\mid$	v	(value)	$\mid$	$\nu x.t$	(fresh)
	$\mid$	$t_1 t_2$	(application)	$\mid$	$t_1 := t_2$	(write)
	$\mid$	$\mathtt{fix}(t)$	(fixed point)	$\mid$	$t_1 \cap t_2$	(intersection)
	$\mid$	$t_1 <_? t_2$	(less than)	$\mid$	$t_1 =_? t_2$	(equality)
	$\mid$	let $x, y = t_1$ in t_2	(pair destr.)	$\mid$	let $x = t_1$ in t_2	(cbv)
	$\mid$	$t_1 + t_2$	(addition)			

The computational system $TT_{\mathcal{C}}^{\square}$ (2)

$v \in$ Val	$::=$	vt	(type)	$\mid$	$\lambda x.t$	(lambda)
	$\mid$	$\underline{n}$	(number)	$\mid$	$\mathtt{inl}(t)$	(left inj.)
	$\mid$	$\langle t_1, t_2 \rangle$	(pair)	$\mid$	$\mathtt{inr}(t)$	(right inj.)
	$\mid$	$\star$	(constant)	$\mid$	δ	(ref. name)
$v \in$ Type	$::=$	$\Pi_{x:t_1} t_2$	(product)	$\mid$	$\mathbb{U}_i$	(universe)
	$\mid$	$\Sigma_{x:t_1} t_2$	(sum)	$\mid$	$t_1 = t_2 \in t$	(equality)
	$\mid$	$\{x : t_1 \mid t_2\}$	(set)	$\mid$	$\|t\|$	(truncation)
	$\mid$	$\mathtt{Nat}$	(naturals)	$\mid$	$t_1 \cap t_2$	(intersection)
	$\mid$	$t_1 + t_2$	(disj. union)	$\mid$	$\mathtt{pure}$	(pure)
$t \in$ Term	$::=$	x	(variable)	$\mid$	$!t$	(read)
	$\mid$	v	(value)	$\mid$	$\nu x.t$	(fresh)
	$\mid$	$t_1 t_2$	(application)	$\mid$	$t_1 := t_2$	(write)
	$\mid$	$\mathtt{fix}(t)$	(fixed point)	$\mid$	$t_1 \cap t_2$	(intersection)
	$\mid$	$t_1 <_? t_2$	(less than)	$\mid$	$t_1 =_? t_2$	(equality)
	$\mid$	let $x, y = t_1$ in t_2	(pair destr.)	$\mid$	let $x = t_1$ in t_2	(cbv)
	$\mid$	$t_1 + t_2$	(addition)			

- A *computational type theory* in the sense of [Con02].

The computational system $TT_{\mathcal{C}}^{\square}$ (2)

$v \in$ Val	$::=$	vt	(type)	$\mid$	$\lambda x.t$	(lambda)
	$\mid$	$\underline{n}$	(number)	$\mid$	$\mathtt{inl}(t)$	(left inj.)
	$\mid$	$\langle t_1, t_2 \rangle$	(pair)	$\mid$	$\mathtt{inr}(t)$	(right inj.)
	$\mid$	$\star$	(constant)	$\mid$	δ	(ref. name)
$v \in$ Type	$::=$	$\Pi_{x:t_1} t_2$	(product)	$\mid$	$\mathbb{U}_i$	(universe)
	$\mid$	$\Sigma_{x:t_1} t_2$	(sum)	$\mid$	$t_1 = t_2 \in t$	(equality)
	$\mid$	$\{x : t_1 \mid t_2\}$	(set)	$\mid$	$\lVert t \rVert$	(truncation)
	$\mid$	$\mathtt{Nat}$	(naturals)	$\mid$	$t_1 \cap t_2$	(intersection)
	$\mid$	$t_1 + t_2$	(disj. union)	$\mid$	$\mathtt{pure}$	(pure)
$t \in$ Term	$::=$	x	(variable)	$\mid$	$!t$	(read)
	$\mid$	v	(value)	$\mid$	$\nu x.t$	(fresh)
	$\mid$	$t_1 t_2$	(application)	$\mid$	$t_1 := t_2$	(write)
	$\mid$	$\mathtt{fix}(t)$	(fixed point)	$\mid$	$t_1 \cap t_2$	(intersection)
	$\mid$	$t_1 <_? t_2$	(less than)	$\mid$	$t_1 =_? t_2$	(equality)
	$\mid$	let $x, y = t_1$ in t_2	(pair destr.)	$\mid$	let $x = t_1$ in t_2	(cbv)
	$\mid$	$t_1 + t_2$	(addition)			

- A *computational type theory* in the sense of [Con02].
- Typing is extrinsic.

Implementing **ICP** in $TT_{\mathcal{C}}^{\square}$ (1)

Our $TT_{\mathcal{C}}^{\square}$ program, expressed in OCaml[2].

```
type baire = nat -> nat

type brouwer_tree = Leaf of nat | Branch of (nat -> brouwer_tree)

let m : nat ref = ref 0

let generic (ns : nat list) : nat -> nat = fun i ->
  m := max i !m;
  if i >= List.length ns then 0 else List.nth ns i

let compute_btree (f : baire -> nat) : brouwer_tree =
  let rec loop (ns : nat list) : brouwer_tree =
    let i = f (generic ns) in
    if !m < List.length ns then
      Leaf i
    else
      Branch (fun n -> loop (ns @ [n]))
  in loop []
```

[2] Our presentation of the program here follows Sterling [Ste21].

Implementing **ICP** in $\mathrm{TT}_{\mathcal{C}}^{\square}$ (2)

We can now define the function `follow` that decodes the computation encoded by the path given by α.

```
let follow (alpha : baire) : brouwer_tree -> nat =
  let rec loop (n : nat) (t : brouwer_tree) : nat =
    match t with
    | Leaf    k   -> k
    | Branch phi -> loop (1 + n) (phi (alpha n))
  in loop 0
```

The modulus at α is then just the depth of the path given by α.

```
let modulus_at (alpha : baire) : brouwer_tree -> nat =
  let rec loop (n : nat) (t : brouwer_tree) : nat =
    match t with
    | Leaf    _   -> n
    | Branch phi -> loop (1 + n) (phi (alpha n))
  in loop 0
```

An overview of the proof

Goal: Our $\mathsf{TT}^{\square}_{\mathcal{C}}$ implementation of the aforementioned program inhabits the type:

$$\Pi^{\mathsf{p}}_{F:\mathfrak{B}\to\mathtt{Nat}} \left\|\Sigma_{d:\mathsf{BTree}}\,\Pi^{\mathsf{p}}_{\alpha:\mathfrak{B}}\,\mathtt{follow}(d,\alpha) = F(\alpha)\right\|.$$

- **Step 1**: We start with a version of the program that gives a Brouwer co-tree.

An overview of the proof

Goal: Our $\mathsf{TT}^{\square}_{\mathcal{C}}$ implementation of the aforementioned program inhabits the type:

$$\Pi^{\mathsf{p}}_{F:\mathfrak{B}\to\mathtt{Nat}} \left\| \Sigma_{d:\mathsf{BTree}} \Pi^{\mathsf{p}}_{\alpha:\mathfrak{B}} \mathtt{follow}(d,\alpha) = F(\alpha) \right\|.$$

- **Step 1**: We start with a version of the program that gives a Brouwer co-tree.
- **Step 2**: Given a $F : \mathfrak{B} \to \mathtt{Nat}$, we compute the Brouwer co-tree and proceed by case analysis (using classical logic) on whether the co-tree contains an infinite path or not.

An overview of the proof

Goal: Our $\mathsf{TT}^{\square}_{\mathcal{C}}$ implementation of the aforementioned program inhabits the type:

$$\Pi^{\mathsf{p}}_{F:\mathfrak{B}\to\mathtt{Nat}} \left\| \Sigma_{d:\mathsf{BTree}} \Pi^{\mathsf{p}}_{\alpha:\mathfrak{B}} \mathtt{follow}(d,\alpha) = F(\alpha) \right\|.$$

- **Step 1**: We start with a version of the program that gives a Brouwer co-tree.
- **Step 2**: Given a $F : \mathfrak{B} \to \mathtt{Nat}$, we compute the Brouwer co-tree and proceed by case analysis (using classical logic) on whether the co-tree contains an infinite path or not.
  - **Step 3**: Existence of an infinite path contradicts the continuity of F.

An overview of the proof

Goal: Our $\mathsf{TT}^{\square}_{\mathcal{C}}$ implementation of the aforementioned program inhabits the type:

$$\Pi^{\mathsf{p}}_{F:\mathfrak{B}\to\mathtt{Nat}} \left\|\Sigma_{d:\mathsf{BTree}}\, \Pi^{\mathsf{p}}_{\alpha:\mathfrak{B}}\, \mathtt{follow}(d,\alpha) = F(\alpha)\right\|.$$

- **Step 1**: We start with a version of the program that gives a Brouwer co-tree.
- **Step 2**: Given a $F : \mathfrak{B} \to \mathtt{Nat}$, we compute the Brouwer co-tree and proceed by case analysis (using classical logic) on whether the co-tree contains an infinite path or not.
  - **Step 3**: Existence of an infinite path contradicts the continuity of F.
  - **Step 4**: In the case where all the branches of t are finite, we transform the Brouwer co-tree into a Brouwer tree.

An overview of the proof

Goal: Our $\mathsf{TT}^{\square}_{\mathcal{C}}$ implementation of the aforementioned program inhabits the type:

$$\Pi^{\mathsf{p}}_{F:\mathfrak{B}\to\mathtt{Nat}} \left\|\Sigma_{d:\mathsf{BTree}}\, \Pi^{\mathsf{p}}_{\alpha:\mathfrak{B}}\, \mathtt{follow}(d,\alpha) = F(\alpha)\right\|.$$

- **Step 1**: We start with a version of the program that gives a Brouwer co-tree.
- **Step 2**: Given a $F : \mathfrak{B} \to \mathtt{Nat}$, we compute the Brouwer co-tree and proceed by case analysis (using classical logic) on whether the co-tree contains an infinite path or not.
  - **Step 3**: Existence of an infinite path contradicts the continuity of F.
  - **Step 4**: In the case where all the branches of t are finite, we transform the Brouwer co-tree into a Brouwer tree.
- **Step 5**: We then show that the resulting Brouwer tree d satisfies the desired property of $\mathtt{follow}(d,\alpha) = F(\alpha)$.

Deriving **UCP**

Brouwer proved [Bee80; Bro27] that all real-valued functions on the unit interval are uniformly continuous using **Cont** and his Fan Theorem, which he derived from his Bar Thesis.

In our case, **ICP** is strong enough to give **UCP** without the Fan Theorem.

Key idea: if $\mathfrak{B} \rightarrow \texttt{Nat}$ is restricted to $\mathfrak{C} \rightarrow \texttt{Nat}$, the modulus of uniform continuity is the depth of the longest path, which can be computed independently of the input.

Conclusion and further work

We have constructed a program in $\mathsf{TT}_{\mathcal{C}}^{\Box}$ that realises **ICP** (for pure functions), by making use of references.

[3] Code available at `https://github.com/vrahli/opentt`.

Conclusion and further work

We have constructed a program in $\mathsf{TT}_{\mathcal{C}}^{\square}$ that realises **ICP** (for pure functions), by making use of references.

Our results are completely formalised in the Agda proof assistant[3].

[3] Code available at `https://github.com/vrahli/opentt`.

Conclusion and further work

We have constructed a program in $\mathsf{TT}_{\mathcal{C}}^{\square}$ that realises **ICP** (for pure functions), by making use of references.

Our results are completely formalised in the Agda proof assistant[3].

Some further questions to investigate:

- Can we generalise references to more general effects?

[3] Code available at `https://github.com/vrahli/opentt`.

Conclusion and further work

We have constructed a program in $\mathsf{TT}^{\square}_{\mathcal{C}}$ that realises **ICP** (for pure functions), by making use of references.

Our results are completely formalised in the Agda proof assistant[3].

Some further questions to investigate:

- Can we generalise references to more general effects?
- We have not yet shown that **Cont** is strictly weaker than **ICP**.

[3] Code available at `https://github.com/vrahli/opentt`.

Conclusion and further work

We have constructed a program in $\mathsf{TT}^{\square}_{\mathcal{C}}$ that realises **ICP** (for pure functions), by making use of references.

Our results are completely formalised in the Agda proof assistant[3].

Some further questions to investigate:

- Can we generalise references to more general effects?
- We have not yet shown that **Cont** is strictly weaker than **ICP**.
- *Big question*: can we make this (or possibly a different) program work for all $\mathsf{TT}^{\square}_{\mathcal{C}}$ functions instead of just the pure ones?

[3] Code available at `https://github.com/vrahli/opentt`.

References I

[Bee80] Michael J. Beeson. *Foundations of Constructive Mathematics*. Berlin, Heidelberg: Springer Verlag, 1980.

[BMP22] Martin Baillon, Assia Mahboubi, and Pierre-Marie Pédrot. “Gardening with the Pythia A Model of Continuity in a Dependent Setting”. In: ed. by Florin Manea and Alex Simpson. Vol. 216. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 5:1–5:18. DOI: `10.4230/LIPIcs.CSL.2022.5`. URL: `https://doi.org/10.4230/LIPIcs.CSL.2022.5`.

[Bro27] Luitzen E. J. Brouwer. “Über Definitionsbereiche von Funktionen”. In: *Mathematische annalen* 97 (1927), pp. 60–75.

References II

[CJ12] Thierry Coquand and Guilhem Jaber. “A Computational Interpretation of Forcing in Type Theory”. In: *Epistemology versus Ontology*. Ed. by Peter Dybjer et al. Vol. 27. Logic, Epistemology, and the Unity of Science. Springer Netherlands, 2012, pp. 203–213. ISBN: 978-94-007-4434-9.

[Con02] Robert L. Constable. “Naïve Computational Type Theory”. In: *Proof and System-Reliability*. Ed. by Helmut Schwichtenberg and Ralf Steinbrüggen. Dordrecht: Springer Netherlands, 2002, pp. 213–259. ISBN: 978-94-010-0413-8. DOI: 10.1007/978-94-010-0413-8_7. URL: https://doi.org/10.1007/978-94-010-0413-8_7.

References III

[CR22] Liron Cohen and Vincent Rahli. “Constructing Unprejudiced Extensional Type Theories with Choices via Modalities”. In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Ed. by Amy P. Felty. Vol. 228. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 10:1–10:23. ISBN: 978-3-95977-233-4. DOI: 10.4230/LIPIcs.FSCD.2022.10. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16291.

[CR23] Liron Cohen and Vincent Rahli. *$TT_{\mathcal{C}}^{\square}$: a Family of Extensional Type Theories with Effectful Realizers of Continuity*. 2023. arXiv: 2307.14168 [cs.LO].

References IV

[Esc13] Martín H. Escardó. “Continuity of Gödel’s System T Definable Functionals via Effectful Forcing”. In: *Electronic Notes in Theoretical Computer Science*. Vol. 298. Elsevier Science Publishers B. V., 2013, pp. 119–141.

[GHP06] Neil Ghani, Peter G. Hancock, and Dirk Pattinson. “Continuous Functions on Final Coalgebras”. In: *CMCS*. Ed. by Neil Ghani and John Power. Vol. 164. Electronic Notes in Theoretical Computer Science 1. Elsevier Science Publishers B. V., 2006, pp. 141–155. DOI: `10.1016/j.entcs.2006.06.009`. URL: `https://doi.org/10.1016/j.entcs.2006.06.009`.

References V

[Lon99] John Longley. “When is a Functional Program Not a Functional Program?” In: *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’99. Paris, France: Association for Computing Machinery, 1999, pp. 1–7. ISBN: 1581131119. DOI: 10.1145/317636.317775. URL: https://doi.org/10.1145/317636.317775.

[RB18] Vincent Rahli and Mark Bickford. “Validating Brouwer’s continuity principle for numbers using named exceptions”. In: *Mathematical Structures in Computer Science* 28.6 (2018), pp. 942–990. DOI: 10.1017/S0960129517000172.

References VI

[Ste21] Jonathan Sterling. "Higher order functions and Brouwer's thesis". In: *Journal of Functional Programming* 31 (2021). *Bob Harper Festschrift Collection*, e11. DOI: 10.1017/S0956796821000095. arXiv: 1608.03814 [math.LO].