Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 298 (2013) 119–141

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Continuity of Gödel's System T Definable Functionals via Effectful Forcing

Martín Escardó[1]

School of Computer Science
University of Birmingham
Birmingham, England

Abstract

It is well-known that the Gödel's system T definable functions $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ are continuous, and that their restrictions from the Baire type $(\mathbb{N} \to \mathbb{N})$ to the Cantor type $(\mathbb{N} \to 2)$ are uniformly continuous. We offer a new, relatively short and self-contained proof. The main technical idea is a concrete notion of generic element that doesn't rely on forcing, Kripke semantics or sheaves, which seems to be related to generic effects in programming. The proof uses standard techniques from programming language semantics, such as dialogues, monads, and logical relations. We write this proof in intensional Martin-Löf type theory (MLTT) from scratch, in Agda notation. Because MLTT has a computational interpretation and Agda can be seen as a programming language, we can run our proof to compute moduli of (uniform) continuity of T-definable functions.

Keywords: Gödel's system T, continuity, uniform continuity, Baire space, Cantor space, intensional Martin-Löf theory, Agda, dialogue, semantics, logical relation.

1 Introduction

This is a relatively short, and self-contained, proof of the well-known fact that any function $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ that is definable in Gödel's system T is continuous, and that its restriction from the Baire type $(\mathbb{N} \to \mathbb{N})$ to the Cantor type $(\mathbb{N} \to 2)$ is uniformly continuous [15,2]. We believe the proof is new, although it is related to previous work discussed below. The main technical idea is a concrete notion of generic element that doesn't rely on forcing, Kripke semantics or sheaves, which seems to be related to generic effects in programming [13]. Several well-known ideas from logic, computation, constructive mathematics and programming-language semantics naturally appear here, in a relatively simple, self-contained, and hopefully appealing, development.

[1] Email: m.escardo@cs.bham.ac.uk

http://dx.doi.org/10.1016/j.entcs.2013.09.010

The idea is to represent a function $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ by a well-founded dialogue tree, and extract continuity information about f from this tree. To calculate such a tree from a system T term t: $(\iota \Rightarrow \iota) \Rightarrow \iota$ denoting f, we work with an auxiliary interpretation of system T, which gives a function $\tilde{f} : (\tilde{\mathbb{N}} \to \tilde{\mathbb{N}}) \to \tilde{\mathbb{N}}$, where $\tilde{\mathbb{N}}$ is the set of dialogue trees. Applying $\tilde{f}$ to a certain *generic sequence* $\tilde{\mathbb{N}} \to \tilde{\mathbb{N}}$, the desired dialogue tree is obtained. We now explain this idea in more detail.

In the set-theoretical model of system T, the ground type ι is interpreted as the set $\mathbb{N}$ of natural numbers, and if the types σ and τ are interpreted as sets X and Y, then the type $\sigma \Rightarrow \tau$ is interpreted as the set of all functions $X \to Y$. We consider an auxiliary model that replaces the interpretation of the ground type by the set $\tilde{\mathbb{N}}$, but keeps the interpretation of $\Rightarrow$ as the formation of the set of all functions. In this model, the zero constant is interpreted by a suitable element $\tilde{0}$ of $\tilde{\mathbb{N}}$, the successor constant is interpreted by a function $\tilde{\mathbb{N}} \to \tilde{\mathbb{N}}$, and each iteration combinator is interpreted by a function $(X \to X) \to X \to \tilde{\mathbb{N}} \to X$. An element of the set $\tilde{\mathbb{N}}$ is a well-founded dialogue tree that describes the computation of a natural number relative to an unspecified oracle $\alpha : \mathbb{N} \to \mathbb{N}$. An internal node is labeled by a natural number representing a query to the oracle, and has countably many branches corresponding to the possible answers. Each leaf is labeled by a natural number and represents a possible outcome of the computation. These dialogues represent computations in the sense of Kleene [10].

If a particular oracle $\alpha : \mathbb{N} \to \mathbb{N}$ is given, we get a natural number from any $d \in \tilde{\mathbb{N}}$ via a decodification function

$$\text{decode} : (\mathbb{N} \to \mathbb{N}) \to \tilde{\mathbb{N}} \to \mathbb{N}.$$

It turns out that there is a function

$$\text{generic} : \tilde{\mathbb{N}} \to \tilde{\mathbb{N}}$$

that can be regarded as a *generic sequence* in the sense that, for any particular sequence $\alpha : \mathbb{N} \to \mathbb{N}$,

$$\begin{tikzcd}
\tilde{\mathbb{N}} \arrow[r, "\text{generic}"] \arrow[d, "\text{decode } \alpha"'] & \tilde{\mathbb{N}} \arrow[d, "\text{decode } \alpha"] \\
\mathbb{N} \arrow[r, "\alpha"'] & \mathbb{N}.
\end{tikzcd}$$

That is, the generic sequence codes any concrete sequence α, provided the sequence α itself is used as the concrete oracle for decodification. The idea is that the application of the function generic to a dialogue tree adds a new layer of choices at its leaves.

Next we show that for any given term $t : (\iota \Rightarrow \iota) \Rightarrow \iota$ denoting a function $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ in the standard interpretation and $\tilde{f} : (\tilde{\mathbb{N}} \to \tilde{\mathbb{N}}) \to \tilde{\mathbb{N}}$ in the dialogue interpretation, we have that

$$f\ \alpha = \text{decode}\ \alpha\ (\tilde{f}\ \text{generic}).$$

This is proved by establishing a logical relation between the set-theoretic and dialogue models. Thus we can compute a dialogue tree of f by applying $\tilde{f}$ to the generic sequence.

The set $\tilde{\mathbb{N}}$ is constructed as $\mathrm{B}\ \mathbb{N}$ for a suitable dialogue monad B. Then the interpretation of the constant zero is $\eta\ 0$ where η is the unit of the monad, the interpretation of the successor constant is given by functoriality as B succ, and the interpretation of the primitive recursion constant is given by the Kleisli extension of its standard interpretation. The object part $\mathrm{B}\ X$ of the monad is inductively defined by the constructors

$\eta : X \to \mathrm{B}\ X,$
$\beta : (\mathbb{N} \to \mathrm{B}\ X) \to \mathbb{N} \to \mathrm{B}\ X,$

where η constructs leaves and β constructs a tree $\beta\ \phi\ n$ given countably many trees ϕ and a label n. With $X = \mathbb{N}$, we have $\beta\ \eta : \mathbb{N} \to \mathrm{B}\ \mathbb{N}$, and the generic sequence is the Kleisli extension of $\beta\ \eta$. Thus, the generic sequence seems to be a sort of *generic effect* in the sense of [13]. Notice that our interpretation is a call-by-name version of Moggi's semantics.

Using this, it follows that if a function $f : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ is the set-theoretical interpretation of some system T term $t : (\iota \Rightarrow \iota) \Rightarrow \iota$, then it is continuous and its restriction to $\mathbb{N} \to 2$ is uniformly continuous, where 2 is the set with elements 0 and 1. The reason is that a dialogue produces an answer after finitely many queries, because it is well-founded, and that a dialogue tree for a function $(\mathbb{N} \to 2) \to \mathbb{N}$ is finite, because it is finitely branching. Recall that continuity means that, for any sequence of integers $\alpha : \mathbb{N} \to \mathbb{N}$, there is $m : \mathbb{N}$, called a modulus of continuity of f at the point α, such that any sequence α' that agrees with α at the first m positions gives the same result, that is, $f\ \alpha = f\ \alpha'$. Uniform continuity means that there is $m : \mathbb{N}$, called a modulus of uniform continuity of f on $\mathbb{N} \to 2$, such that any two binary sequences α and α' that agree at the first m positions give the same result.

Our arguments are constructive, and we write the full proof from scratch in intensional Martin-Löf type theory (MLTT), in Agda notation [4], without the use of libraries. We don't assume previous familiarity with Agda, but we do require rudimentary knowledge of MLTT. The Agda source file for this program/proof [7] is written in Knuth's *literate* style, which automatically generates the LaTeX file that produces this article. Agda both checks proofs and can run them. Notice that MLTT or Agda cannot prove or disprove that all functions $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ are continuous, as they are compatible with both classical and constructive mathematics, like Bishop mathematics [3]. The theorem here is that certain functions $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ are continuous: those that can be defined in system T.

Related work. The idea of computing continuity information by applying a function to effectful arguments goes back to Longley [11], who passes exceptions to the function. A similar approach is described in an example given by Bauer and Pretnar [1].

The idea of working with computation trees is of course very old, going back to Brouwer [2] in intuitionistic mathematics, and to Kleene [10] in computability theory in the form of dialogues, where the input is referred to as an oracle. Howard [9] derives computation trees for system T, by operational methods, by successively reducing a term so that each time an oracle given by a free variable of type $\iota \Rightarrow \iota$ is queried, countably many branches of the computation are created, corresponding to the possible answers given by the oracle. Hancock and Setzer use variations of dialogue trees to describe interactive computation in type theory [12] (see also [8]).

Our work is directly inspired by Coquand and Jaber's work on forcing in type theory [5,6]. Like Howard, they derive computation trees by operational methods. They extend dependent type theory with a constant for a generic element, and then decorate judgements with subscripts that keep track of approximation information about the generic element as the computations proceed (similarly to [15]). In this way they extract continuity information. They prove the termination and soundness of this modification of type theory using Tait's computability method, which here is manifested as a logical relation between our two models. They also provide a Haskell implementation for the system T case as an appendix, which uses a monad that is the composition of the list monad (for nondeterminism) and of the state monad. Their Haskell program implements a normalization procedure with bookkeeping information, tracked by the monad, that produces computation trees. Because they only account for uniform continuity in their Haskell implementation, such trees are finite. They describe their work as a computational interpretation of forcing and continuity as presented in Beeson [2]. The difference is that their approach is syntactical whereas ours is semantical, and the reader may sense an analogy with normalization by evaluation. Notice that these arguments only show that the *definable* functions are continuous. To get a constructive model in which *all* functions are continuous, they work with iterated forcing, which is related to our recent work [16], but this is another story.

Organization. (2) Formal proof in Agda. (3) Informal, rigorous proof.

2 Proof in Martin-Löf type theory in Agda notation

2.1 Agda preliminaries

The purpose of this subsection is two-fold: (1) To develop a tiny Agda library for use in the following sections, and (2) to briefly explain Agda notation [4] for MLTT. We assume rudimentary knowledge of (intensional) Martin-Löf type theory and the BHK interpretation of the quantifiers as products Π and sums Σ. We don't use any feature of Agda that goes beyond standard MLTT. If we were trying to be purist, we would use W-types rather than some of our inductive definitions using the Agda keyword *data*. Notice that the coloured text in the electronic version of this paper is the Agda code.

The universe of all types is denoted by Set, and types are called sets (this is a universe à la Russell). Products Π are denoted by ∀ in Agda. Consider the definition of the (interpretation of) the standard combinators:

```
Ķ : ∀{X Y : Set} → X → Y → X
Ķ x y = x
Ş : ∀{X Y Z : Set} → (X → Y → Z) → (X → Y) → X → Z
Ş f g x = f x (g x)
```

The curly braces around the set variables indicate that these are implicit parameters, to be inferred by Agda whenever Ķ and Ş are used. If Agda fails to uniquely infer the missing arguments, one has to write e.g. Ķ $\{X\}$ $\{Y\}$ x y rather than the abbreviated form Ķ x y. The following should be self-explanatory:

```
_∘_ : ∀{X Y Z : Set} → (Y → Z) → (X → Y) → (X → Z)
g ∘ f = λ x → g(f x)

data ℕ : Set where
    zero : ℕ
    succ : ℕ → ℕ
rec : ∀{X : Set} → (X → X) → X → ℕ → X
rec f x    zero    = x
rec f x (succ n) = f(rec f x n)
```

Agda has a termination checker that verifies that recursions are well-founded, and hence all functions are total. We also need types of binary digits, finite lists, and finite binary trees:

```
data ℕ₂ : Set where
    ₀ ₁ : ℕ₂

data List (X : Set) : Set where
    []    : List X
    _::_ : X → List X → List X

data Tree (X : Set) : Set where
    empty : Tree X
    branch : X → (ℕ₂ → Tree X) → Tree X
```

Sums are not built-in and hence need to be defined:

```
data Σ {X : Set} (Y : X → Set) : Set where
    _,_ : ∀(x : X)(y : Y x) → Σ {X} Y
```

The definition says that an element of Σ $\{X\}$ Y is a pair $(x{,}y)$ with $x : X$ and $y : Y\ x$. Notice that comma is not a reserved symbol: we define it as a binary operator to construct dependent pairs. Because $Y = \lambda(x : X) \to Y\ x$ if one assumes the η-law, and because the first argument is implicit, we can write Σ $\{X\}$ Y as Σ Y or Σ $\backslash(x : X) \to Y\ x$, where backslash is the same thing as lambda. We will use backslash exclusively for sums.

```
π₀ : ∀{X : Set} {Y : X → Set} → (Σ \(x : X) → Y x) → X
π₀(x , y) = x
π₁ : ∀{X : Set} {Y : X → Set} → ∀(t : Σ \(x : X) → Y x) → Y(π₀ t)
π₁(x , y) = y
```

The identity type Id X x y is written $x \equiv y$ with X implicit, and is inductively defined as "the least reflexive relation":

```
data _≡_ {X : Set} : X → X → Set where
    refl : ∀{x : X} → x ≡ x

sym : ∀{X : Set} → ∀{x y : X} → x ≡ y → y ≡ x
sym refl = refl
trans : ∀{X : Set} → ∀{x y z : X} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
cong : ∀{X Y : Set} → ∀(f : X → Y) → ∀{x₀ x₁ : X} → x₀ ≡ x₁ → f x₀ ≡ f x₁
cong f refl = refl
cong₂ : ∀{X Y Z : Set} → ∀(f : X → Y → Z)
```

```
    → ∀{x₀ x₁ : X}{y₀ y₁ : Y} → x₀ ≡ x₁ → y₀ ≡ y₁ → f x₀ y₀ ≡ f x₁ y₁
cong₂ f refl refl = refl
```

2.2 Dialogues and continuity

We consider the computation of functionals $(X \to Y) \to Z$ with dialogue trees. We work with the following inductively defined type of (well founded) dialogue trees indexed by three types X, Y and Z. These are Y-branching trees with X-labeled internal nodes and Z-labeled leaves:

```
data D (X Y Z : Set) : Set where
  η : Z → D X Y Z
  β : (Y → D X Y Z) → X → D X Y Z
```

A leaf is written $\eta\ z$, and it gives the final answer z (η will be the unit of a monad). A forest is a Y-indexed family ϕ of trees. Given such a forest ϕ and $x : X$, we can build a new tree $\beta\ \phi\ x$ whose root is labeled by x, which has a subtree $\phi\ y$ for each $y : Y$. We can imagine $x : X$ as query, for which an oracle α gives some intermediate answer $y = \alpha\ x$: Y. After this answer y, we move to the subtree $\phi\ y$, and the dialogue proceeds in this way, until a leaf with the final answer is reached:

```
dialogue : ∀{X Y Z : Set} → D X Y Z → (X → Y) → Z
dialogue (η z)     α = z
dialogue (β φ x) α = dialogue (φ(α x)) α
```

We say that a function $(X \to Y) \to Z$ is *eloquent* if it is computed by some dialogue:

```
eloquent : ∀{X Y Z : Set} → ((X → Y) → Z) → Set
eloquent f = Σ \d → ∀ α → dialogue d α ≡ f α
```

Here we are interested in the case $X{=}Y{=}Z{=}\mathbb{N}$. Think of functions $\alpha : \mathbb{N} \to \mathbb{N}$ as sequences of natural numbers. The set of such sequences is called the Baire space:

```
Baire : Set
Baire = ℕ → ℕ
```

Functions Baire $\to \mathbb{N}$ are coded by a particular kind of dialogue trees, namely B $\mathbb{N}$ where B is defined as follows:

```
B : Set → Set
B = D ℕ ℕ
```

We work with a refined version of continuity, which gives more information than the traditional notion introduced in Section 1, where the modulus of continuity is a finite list of indices rather than an upper bound for the indices. The agreement relation determined by a list of indices is inductively defined as follows, where $\alpha \equiv[\ s\ ]\ \alpha'$ says that the sequences α and α' agree at the indices collected in the list s:

(i) $\alpha \equiv[\ []\ ]\ \alpha'$,

(ii) $\alpha\ i \equiv \alpha'\ i \to \alpha \equiv[\ s\ ]\ \alpha' \to \alpha \equiv[\ i :: s\ ]\ \alpha'$.

We write this inductive definition as follows in Agda, where we give the name [] to the proof of the first clause and the name :: to the proof of the second clause, that is, using the same constructor names as for the inductively defined type of lists:

```
data _≡[_]_ {X : Set} : (ℕ → X) → List ℕ → (ℕ → X) → Set where
  []   : ∀{α α' : ℕ → X} → α ≡[ [] ] α'
  _::_ : ∀{α α' : ℕ → X}{i : ℕ}{s : List ℕ} → α i ≡ α' i → α ≡[ s ] α' → α ≡[ i :: s ] α'

continuous : (Baire → ℕ) → Set
continuous f = ∀(α : Baire) → Σ \(s : List ℕ) → ∀(α' : Baire) → α ≡[ s ] α' → f α ≡ f α'
```

It is an easy exercise, left to the reader, to produce an Agda proof that this refined notion of continuity implies the traditional notion of continuity, by taking the maximum value of the list s. Functions defined by dialogues are continuous, because a dialogue produces an answer after finitely many queries:

```
dialogue-continuity : ∀(d : B ℕ) → continuous(dialogue d)
dialogue-continuity (η n) α = ([] , lemma)
  where
    lemma : ∀ α' → α ≡[ [] ] α' → n ≡ n
    lemma α' r = refl
dialogue-continuity (β φ i) α = ((i :: s) , lemma)
  where
    IH : ∀(i : ℕ) → continuous(dialogue(φ(α i)))
    IH i = dialogue-continuity (φ(α i))
    s : List ℕ
    s = π₀(IH i α)
    claim₀ : ∀(α' : Baire) → α ≡[ s ] α' → dialogue(φ(α i)) α ≡ dialogue(φ(α i)) α'
    claim₀ = π₁(IH i α)
    claim₁ : ∀(α' : Baire) → α i ≡ α' i → dialogue (φ (α i)) α' ≡ dialogue (φ (α' i)) α'
    claim₁ α' r = cong (λ n → dialogue (φ n) α') r
    lemma : ∀(α' : Baire) → α ≡[ i :: s ] α'   → dialogue (φ(α i)) α ≡ dialogue(φ (α' i)) α'
    lemma α' (r :: rs) = trans (claim₀ α' rs) (claim₁ α' r)
```

This formal proof is informally explained as follows. We show that

$$\forall(d : \mathrm{B}\ \mathbb{N}) \to \mathrm{continuous}(\mathrm{dialogue}\ d)$$

by induction on d. Expanding the definition, this amounts to, using Agda notation,

$$\forall\ d \to \forall\ \alpha \to \Sigma\ \backslash s \to \forall\ \alpha' \to \alpha \equiv[\ s\ ]\ \alpha' \to \mathrm{dialogue}\ d\ \alpha \equiv \mathrm{dialogue}\ d\ \alpha'.$$

For the base case $d = \eta\ n$, the definition of the function dialogue gives dialogue $d\ \alpha = n$, and so we must show that, for any α,

$$\Sigma\ \backslash s \to \forall\ \alpha' \to \alpha \equiv[\ s\ ]\ \alpha' \to n \equiv n.$$

We can take $s = []$ and then we are done, because $n \equiv n$ by reflexivity. This is what the first equation of the formal proof says. Thus notice that Agda, in accordance with MLTT, silently expands definitions by reduction to normal form. For the induction step $d = \beta\ \phi\ i$, expanding the definition of the dialogue function, what we need to prove is that, for an arbitrary α,

$$\Sigma\ \backslash s' \to \forall\ \alpha' \to \alpha \equiv[\ s'\ ]\ \alpha' \to \mathrm{dialogue}\ (\phi(\alpha\ i))\ \alpha \equiv \mathrm{dialogue}\ (\phi\ \alpha'\ i)\ \alpha'.$$

The induction hypothesis is $\forall(i : \mathbb{N}) \to \mathrm{continuous}(\mathrm{dialogue}(\phi(\alpha\ i)))$, which gives, for any i and our arbitrary α,

$$\Sigma\ \backslash s \to \forall\ \alpha' \to \alpha \equiv[\ s\ ]\ \alpha' \to \mathrm{dialogue}(\phi(\alpha\ i))\ \alpha = \mathrm{dialogue}(\phi(\alpha\ i))\ \alpha'.$$

Using the two projections π_0 and π_1 we get s and a proof that

$$\forall\ \alpha' \to \alpha \equiv[\ s\ ]\ \alpha' \to \text{dialogue}(\phi(\alpha\ i))\ \alpha = \text{dialogue}(\phi(\alpha\ i))\ \alpha'.$$

Hence we can take $s' = i :: s$, and the desired conclusion holds substituting equals for equals (with cong) using transitivity and the definition $\alpha\ i \equiv \alpha'\ i \to \alpha \equiv[\ s\ ]\ \alpha' \to \alpha \equiv[\ i :: s\ ]\ \alpha'$. This amounts to the second equation of the proof, where in the pattern of the proof of the lemma we have $r : \alpha\ i \equiv \alpha'\ i$ and $rs : \alpha \equiv[\ s\ ]\ \alpha'$.

We need the following technical lemma because it is not provable in intensional MLTT that any two functions are equal if they are pointwise equal. The proof is admitedly written in a rather laconic form. The point is that the notion of continuity depends only on the values of the function, and the hypothesis says that the two functions have the same values. Notice that the axiom of function extensionality (any two pointwise equal functions are equal) is not false but rather not provable or disprovable, and is consistent [14].

```
continuity-extensional : ∀(f g : Baire → ℕ) → (∀ α → f α ≡ g α) → continuous f → continuous g
continuity-extensional f g t c α = (π₀(c α) , (λ α' r → trans (sym (t α)) (trans (π₁(c α) α' r) (t α'))))
eloquent-is-continuous : ∀(f : Baire → ℕ) → eloquent f → continuous f
eloquent-is-continuous f (d , e) = continuity-extensional (dialogue d) f e (dialogue-continuity d)
```

The development for uniform continuity is similar to the above, with the crucial difference that a dialogue tree in C ℕ is finite:

```
Cantor : Set
Cantor = ℕ → ℕ₂
C : Set → Set
C = D ℕ ℕ₂
```

We work with a refined version of uniform continuity (cf. Section 1), where the modulus is a finite binary tree s of indices rather than an upper bound of the indices. We could have worked with a list of indices, but the proofs are shorter and more direct using trees. The agreement relation defined by a tree of indices is inductively defined as follows, where $\alpha \equiv[[\ s\ ]]\ \alpha'$ says that α and α' agree at the positions collected in the tree s:

```
data _≡[[_]]_ {X : Set} : (ℕ → X) → Tree ℕ → (ℕ → X) → Set where
  empty : ∀{α α' : ℕ → X} → α ≡[[ empty ]] α'
  branch :
    ∀{α α' : ℕ → X}{i : ℕ}{s : ℕ₂ → Tree ℕ}
    → α i ≡ α' i → (∀(j : ℕ₂) → α ≡[[ s j ]] α') → α ≡[[ branch i s ]] α'
```

Again we are using the same constructor names as for the type of trees.

```
uniformly-continuous : (Cantor → ℕ) → Set
uniformly-continuous f = Σ \(s : Tree ℕ) → ∀(α α' : Cantor) → α ≡[[ s ]] α' → f α ≡ f α'

dialogue-UC : ∀(d : C ℕ) → uniformly-continuous(dialogue d)
dialogue-UC (η n) = (empty , λ α α' n → refl)
dialogue-UC (β φ i) = (branch i s , lemma)
  where
  IH : ∀(j : ℕ₂) → uniformly-continuous(dialogue(φ j))
  IH j = dialogue-UC (φ j)
  s : ℕ₂ → Tree ℕ
  s j = π₀(IH j)
  claim : ∀ j α α' → α ≡[[ s j ]] α' → dialogue (φ j) α ≡ dialogue (φ j) α'
  claim j = π₁(IH j)
  lemma : ∀ α α' → α ≡[[ branch i s ]] α' → dialogue (φ (α i)) α ≡ dialogue (φ (α' i)) α'
```

```
    lemma α α' (branch r l) = trans fact₀ fact₁
      where
        fact₀ : dialogue (ϕ (α i)) α ≡ dialogue (ϕ (α' i)) α
        fact₀ = cong (λ j → dialogue(ϕ j) α) r
        fact₁ : dialogue (ϕ (α' i)) α ≡ dialogue (ϕ (α' i)) α'
        fact₁ = claim (α' i) α α' (l(α' i))

UC-extensional : ∀(f g : Cantor → ℕ) → (∀(α : Cantor) → f α ≡ g α)
                   → uniformly-continuous f → uniformly-continuous g
UC-extensional f g t (u , c) = (u , (λ α α' r → trans (sym (t α)) (trans (c α α' r) (t α'))))

eloquent-is-UC : ∀(f : Cantor → ℕ) → eloquent f → uniformly-continuous f
eloquent-is-UC f (d , e) = UC-extensional (dialogue d) f e (dialogue-UC d)
```

We finish this section by showing that the restriction of an eloquent function f : Baire → ℕ to the Cantor type is also eloquent. We first define a pruning function from B ℕ to C ℕ that implements restriction:

```
embed-ℕ₂-ℕ : ℕ₂ → ℕ
embed-ℕ₂-ℕ ₀ = zero
embed-ℕ₂-ℕ ₁ = succ zero

embed-C-B : Cantor → Baire
embed-C-B α = embed-ℕ₂-ℕ ∘ α

C-restriction : (Baire → ℕ) → (Cantor → ℕ)
C-restriction f = f ∘ embed-C-B

prune : B ℕ → C ℕ
prune (η n) = η n
prune (β ϕ i) = β (λ j → prune(ϕ(embed-ℕ₂-ℕ j))) i

prune-behaviour : ∀(d : B ℕ)(α : Cantor) → dialogue (prune d) α ≡ C-restriction(dialogue d) α
prune-behaviour (η n)    α = refl
prune-behaviour (β ϕ n) α = prune-behaviour (ϕ(embed-ℕ₂-ℕ(α n))) α

eloquent-restriction : ∀(f : Baire → ℕ) → eloquent f → eloquent(C-restriction f)
eloquent-restriction f (d , c) = (prune d , λ α → trans (prune-behaviour d α) (c (embed-C-B α)))
```

2.3 *Gödel's system T extended with an oracle*

For simplicity, we work with system T in its original combinatory form. This is no loss of generality, because both the combinatory and the lambda-calculus forms define the same elements of the set-theoretical model, and here we are interested in the continuity of the definable functionals. The system T type expressions and terms are inductively defined as follows:

```
data type : Set where
  ι    : type
  _⇒_ : type → type → type

data T : (σ : type) → Set where
  Zero   : T ι
  Succ   : T(ι ⇒ ι)
  Rec    : ∀{σ : type}     → T((σ ⇒ σ) ⇒ σ ⇒ ι ⇒ σ)
  K      : ∀{σ τ : type}   → T(σ ⇒ τ ⇒ σ)
  S      : ∀{ρ σ τ : type} → T((ρ ⇒ σ ⇒ τ) ⇒ (ρ ⇒ σ) ⇒ ρ ⇒ τ)
  _·_    : ∀{σ τ : type}   → T(σ ⇒ τ) → T σ → T τ

infixr 1 _⇒_
infixl 1 _·_
```

Notice that there are five constants (or combinators) and one binary constructor (application). Notice also that one can build only well-typed terms. The set-theoretical interpretation of type expressions and terms is given by

```
Set⟦_⟧ : type → Set
Set⟦ ι ⟧ = ℕ
Set⟦ σ ⇒ τ ⟧ = Set⟦ σ ⟧ → Set⟦ τ ⟧


⟦_⟧ : ∀{σ : type} → T σ → Set⟦ σ ⟧
⟦ Zero ⟧    = zero
⟦ Succ ⟧    = succ
⟦ Rec ⟧     = rec
⟦ K ⟧       = Ķ
⟦ S ⟧       = Ş
⟦ t · u ⟧   = ⟦ t ⟧ ⟦ u ⟧
```

An element of the set-theoretical model is called T-definable if there is a T-term denoting it:

```
T-definable : ∀{σ : type} → Set⟦ σ ⟧ → Set
T-definable x = Σ \t → ⟦ t ⟧ ≡ x
```

As discussed above, the main theorem, proved in the last subsection, is that every T-definable function Baire → ℕ is continuous. The system T type of such functionals is $(\iota \Rightarrow \iota) \Rightarrow \iota$.

We also consider system T extended with a formal oracle $\Omega : \iota \Rightarrow \iota$:

```
data TΩ : (σ : type) → Set where
   Ω      : TΩ(ι ⇒ ι)
   Zero   : TΩ ι
   Succ   : TΩ(ι ⇒ ι)
   Rec    : ∀{σ : type}       → TΩ((σ ⇒ σ) ⇒ σ ⇒ ι ⇒ σ)
   K      : ∀{σ τ : type}     → TΩ(σ ⇒ τ ⇒ σ)
   S      : ∀{ρ σ τ : type} → TΩ((ρ ⇒ σ ⇒ τ) ⇒ (ρ ⇒ σ) ⇒ ρ ⇒ τ)
   _·_    : ∀{σ τ : type}     → TΩ(σ ⇒ τ) → TΩ σ → TΩ τ
```

In the standard set-theoretical interpretation, the oracle can be thought of as a free variable ranging over elements of the interpretation Baire of the type expression $\iota \Rightarrow \iota$:

```
⟦_⟧' : ∀{σ : type} → TΩ σ → Baire → Set⟦ σ ⟧
⟦ Ω ⟧'       α = α
⟦ Zero ⟧'    α = zero
⟦ Succ ⟧'    α = succ
⟦ Rec ⟧'     α = rec
⟦ K ⟧'       α = Ķ
⟦ S ⟧'       α = Ş
⟦ t · u ⟧'   α = ⟦ t ⟧' α (⟦ u ⟧' α)
```

To regard TΩ as an extension of T we need to work with an embedding:

```
embed : ∀{σ : type} → T σ → TΩ σ
embed Zero = Zero
embed Succ = Succ
embed Rec = Rec
embed K = K
embed S = S
embed (t · u) = (embed t) · (embed u)
```

2.4 The dialogue interpretation of system T

We now consider an auxiliary interpretation of system T extended with an oracle in order to show that the original T-definable functions Baire → ℕ are continuous. In the alternative semantics, types are interpreted as the underlying objects of certain algebras of the dialogue monad. The ground type is interpreted as the free algebra of the standard interpretation, and function types as function sets. For the sake of brevity, we will include only the parts of the definition of the monad that we actually need for our purposes.

```
kleisli-extension : ∀{X Y : Set} → (X → B Y) → B X → B Y
kleisli-extension f (η x)     = f x
kleisli-extension f (β φ i) = β (λ j → kleisli-extension f (φ j)) i

B-functor : ∀{X Y : Set} → (X → Y) → B X → B Y
B-functor f = kleisli-extension(η ∘ f)
```

The following two lemmas are crucial. We first swap the two arguments of the dialogue function to have the view that from an element of the Baire type we get a B-algebra B $X \to X$ for any X:

```
decode : ∀{X : Set} → Baire → B X → X
decode α d = dialogue d α
```

The decodification map is natural for any oracle α : Baire:

$$\begin{tikzcd}
\mathrm{B}\ X \arrow[r, "\mathrm{B}\ g"] \arrow[d, "\mathrm{decode}\ \alpha"'] & \mathrm{B}\ Y \arrow[d, "\mathrm{decode}\ \alpha"] \\
X \arrow[r, "g"'] & Y.
\end{tikzcd}$$

```
decode-α-is-natural : ∀{X Y : Set}(g : X → Y)(d : B X)(α : Baire) → g(decode α d) ≡ decode α (B-functor g d)
decode-α-is-natural g (η x)     α = refl
decode-α-is-natural g (β φ i) α = decode-α-is-natural g (φ(α i)) α
```

The following diagram commutes for any $f : X \to \mathrm{B}\ Y$:

$$\begin{tikzcd}
\mathrm{B}\ X \arrow[rr, "\text{kleisli-extension}\ f"] \arrow[d, "\mathrm{decode}\ \alpha"'] & & \mathrm{B}\ Y \arrow[d, "\mathrm{decode}\ \alpha"] \\
X \arrow[r, "f"'] & \mathrm{B}\ Y \arrow[r, "\mathrm{decode}\ \alpha"'] & Y.
\end{tikzcd}$$

```
decode-kleisli-extension : ∀{X Y : Set}(f : X → B Y)(d : B X)(α : Baire)
   → decode α (f(decode α d)) ≡ decode α (kleisli-extension f d)
decode-kleisli-extension f (η x)     α = refl
decode-kleisli-extension f (β φ i) α = decode-kleisli-extension f (φ(α i)) α
```

System TΩ type expressions are interpreted as the underlying sets of certain algebras of the dialogue monad. The base type is interpreted as the underlying set of the free algebra of the standard interpretation, and function types are interpreted as sets of functions, exploiting the fact that algebras are exponential ideals (if Y is the underlying object of an algebra, then so is the set of all functions $X \to Y$ for any X, with the pointwise structure).

```
B-Set⟦_⟧ : type → Set
B-Set⟦ ι ⟧ = B(Set⟦ ι ⟧)
B-Set⟦ σ ⇒ τ ⟧ = B-Set⟦ σ ⟧ → B-Set⟦ τ ⟧
```

According to the official definition of an algebra of a monad, to show that a set X is the underlying object of an algebra one must provide a structure map $\mathrm{B}\ X \to X$. Alternatively, which is more convenient for us, one can provide a generalized Kleisli extension operator, defined as follows, where the base case is just Kleisli extension, and the induction step is pointwise extension:

```
Kleisli-extension : ∀{X : Set} {σ : type} → (X → B-Set⟦ σ ⟧) → B X → B-Set⟦ σ ⟧
Kleisli-extension {X} {ι} = kleisli-extension
Kleisli-extension {X} {σ ⇒ τ} = λ g d s → Kleisli-extension {X} {τ} (λ x → g x s) d
```

With this we can now define the dialogue interpretation of system TΩ. The generic element of the Baire type under this interpretation will interpret the Baire oracle Ω:

```
generic : B ℕ → B ℕ
generic = kleisli-extension(β η)
```

As discussed in Section 1, the crucial property of the generic element is this:

$$\begin{tikzcd}
\mathrm{B}\ \mathbb{N} \arrow[r, "\text{generic}"] \arrow[d, "\text{decode } \alpha"'] & \mathrm{B}\ \mathbb{N} \arrow[d, "\text{decode } \alpha"] \\
\mathbb{N} \arrow[r, "\alpha"'] & \mathbb{N}.
\end{tikzcd}$$

```
generic-diagram : ∀(α : Baire)(d : B ℕ) → α(decode α d) ≡ decode α (generic d)
generic-diagram α (η n) = refl
generic-diagram α (β φ n) = generic-diagram α (φ(α n))
```

The alternative interpretations of zero and successor are obvious:

```
zero' : B ℕ
zero' = η zero
succ' : B ℕ → B ℕ
succ' = B-functor succ
```

And the interpretation of the primitive recursion combinator again uses Kleisli extension in an obvious way:

```
rec' : ∀{σ : type} → (B-Set⟦ σ ⟧ → B-Set⟦ σ ⟧) → B-Set⟦ σ ⟧ → B ℕ → B-Set⟦ σ ⟧
rec' f x = Kleisli-extension(rec f x)
```

This gives the dialogue interpretation. Notice that the interpretations of K, S and application are standard. This is because we interpret function types as sets of functions:

```
B⟦_⟧ : ∀{σ : type} → TΩ σ → B-Set⟦ σ ⟧
B⟦ Ω ⟧        = generic
```

```
B⟦ Zero ⟧    = zero'
B⟦ Succ ⟧    = succ'
B⟦ Rec ⟧     = rec'
B⟦ K ⟧       = Ķ
B⟦ S ⟧       = Ş
B⟦ t · u ⟧   = B⟦ t ⟧ (B⟦ u ⟧)
```

This semantics gives the desired dialogue trees:

```
dialogue-tree : T((ι ⇒ ι) ⇒ ι) → B ℕ
dialogue-tree t = B⟦ (embed t) · Ω ⟧
```

The remainder of the development is the formulation and proof of the correctness of the dialogue-tree function. We conclude this section with the trivial proof that the embedding of T into TΩ preserves the standard interpretation and furthermore is independent of oracles:

```
preservation : ∀{σ : type} → ∀(t : T σ) → ∀(α : Baire) → ⟦ t ⟧ ≡ ⟦ embed t ⟧' α
preservation Zero α = refl
preservation Succ α = refl
preservation Rec α = refl
preservation K α = refl
preservation S α = refl
preservation (t · u) α = cong₂ (λ f x → f x) (preservation t α) (preservation u α)
```

2.5 Relating the two models

The main lemma is that for any term t : TΩ ι,

$$⟦ t ⟧' \alpha \equiv \text{decode } \alpha \ (\mathrm{B}⟦ t ⟧).$$

We use the following logical relation to prove this:

```
R : ∀{σ : type} → (Baire → Set⟦ σ ⟧) → B-Set⟦ σ ⟧ → Set

R {ι} n n' =
   ∀(α : Baire) → n α ≡ decode α n'

R {σ ⇒ τ} f f' =
   ∀(x : Baire → Set⟦ σ ⟧)(x' : B-Set⟦ σ ⟧) → R {σ} x x' → R {τ} (λ α → f α (x α)) (f' x')
```

We need a (fairly general) technical lemma, which is used for constants with an interpretation using the Kleisli-extension operator. In our case, this is just the recursion combinator. The proof is by induction on type expressions, crucially relying on the lemma *decode-kleisli-extension*, but is routine otherwise:

```
R-kleisli-lemma : ∀(σ : type)(g : ℕ → Baire → Set⟦ σ ⟧)(g' : ℕ → B-Set⟦ σ ⟧)
   → (∀(k : ℕ) → R (g k) (g' k))
   → ∀(n : Baire → ℕ)(n' : B ℕ) → R n n' → R (λ α → g (n α) α) (Kleisli-extension g' n')

R-kleisli-lemma ι g g' rg n n' rn = λ α → trans (fact₃ α) (fact₀ α)
   where
      fact₀ : ∀ α → decode α (g' (decode α n')) ≡ decode α (kleisli-extension g' n')
      fact₀ = decode-kleisli-extension g' n'
      fact₁ : ∀ α → g (n α) α ≡ decode α (g'(n α))
      fact₁ α = rg (n α) α
      fact₂ : ∀ α → decode α (g' (n α)) ≡ decode α (g' (decode α n'))
      fact₂ α = cong (λ k → decode α (g' k)) (rn α)
      fact₃ : ∀ α → g (n α) α ≡ decode α (g' (decode α n'))
      fact₃ α = trans (fact₁ α) (fact₂ α)
```

```
R-kleisli-lemma (σ ⇒ τ) g g' rg n n' rn
  = λ y y' ry → R-kleisli-lemma τ (λ k α → g k α (y α)) (λ k → g' k y') (λ k → rg k y y' ry) n n' rn
```

The proof of the main lemma is by induction on terms, crucially relying on the lemmas *generic-diagram* (for the term Ω), *decode-is-natural* (for the term Succ) and *R-kleisli-lemma* (for the term Rec). The terms K and S are routine (but laborious and difficult to get right in an informal calculation), and so is the induction step for application:

```
main-lemma : ∀{σ : type}(t : TΩ σ) → R ⟦ t ⟧' (B⟦ t ⟧)

main-lemma Ω = lemma
  where
    claim : ∀ α n n' → n α ≡ dialogue n' α → α(n α) ≡ α(decode α n')
    claim α n n' s = cong α s
    lemma : ∀(n : Baire → ℕ)(n' : B ℕ) → (∀ α → n α ≡ decode α n')
      → ∀ α → α(n α) ≡ decode α (generic n')
    lemma n n' rn α = trans (claim α n n' (rn α)) (generic-diagram α n')

main-lemma Zero = λ α → refl
main-lemma Succ = lemma
  where
    claim : ∀ α n n' → n α ≡ dialogue n' α → succ(n α) ≡ succ(decode α n')
    claim α n n' s = cong succ s
    lemma : ∀(n : Baire → ℕ)(n' : B ℕ) → (∀ α → n α ≡ decode α n')
      → ∀ α → succ (n α) ≡ decode α (B-functor succ n')
    lemma n n' rn α = trans (claim α n n' (rn α)) (decode-α-is-natural succ n' α)


main-lemma {(σ ⇒ .σ) ⇒ .σ ⇒ ι ⇒ .σ} Rec = lemma
  where
    lemma : ∀(f : Baire → Set⟦ σ ⟧ → Set⟦ σ ⟧)(f' : B-Set⟦ σ ⟧ → B-Set⟦ σ ⟧) → R {σ ⇒ σ} f f'
      → ∀(x : Baire → Set⟦ σ ⟧)(x' : B-Set⟦ σ ⟧)
      → R {σ} x x' → ∀(n : Baire → ℕ)(n' : B ℕ) → R {ι} n n'
      → R {σ} (λ α → rec (f α) (x α) (n α)) (Kleisli-extension(rec f' x') n')
    lemma f f' rf x x' rx = R-kleisli-lemma σ g g' rg
      where
        g : ℕ → Baire → Set⟦ σ ⟧
        g k α = rec (f α) (x α) k
        g' : ℕ → B-Set⟦ σ ⟧
        g' k = rec f' x' k
        rg : ∀(k : ℕ) → R (g k) (g' k)
        rg zero = rx
        rg (succ k) = rf (g k) (g' k) (rg k)

main-lemma K = λ x x' rx y y' ry → rx

main-lemma S = λ f f' rf g g' rg x x' rx → rf x x' rx (λ α → g α (x α)) (g' x') (rg x x' rx)

main-lemma (t · u) = main-lemma t ⟦ u ⟧' B⟦ u ⟧ (main-lemma u)
```

This gives the correctness of the dialogue-tree function defined above: the standard interpretation of a term is computed by its dialogue tree.

```
dialogue-tree-correct : ∀(t : T((ι ⇒ ι) ⇒ ι))(α : Baire) → ⟦ t ⟧ α ≡ decode α (dialogue-tree t)
dialogue-tree-correct t α = trans claim₀ claim₁
  where
  claim₀ : ⟦ t ⟧ α ≡ ⟦ (embed t) · Ω ⟧' α
  claim₀ = cong (λ g → g α) (preservation t α)
  claim₁ : ⟦ (embed t) · Ω ⟧' α ≡ decode α (dialogue-tree t)
  claim₁ = main-lemma ((embed t) · Ω) α
```

The desired result follows directly from this:

```
eloquence-theorem : ∀(f : Baire → ℕ) → T-definable f → eloquent f
eloquence-theorem f (t , r) = (dialogue-tree t , λ α → trans(sym(dialogue-tree-correct t α))(cong(λ g → g α) r))

corollary₀ : ∀(f : Baire → ℕ) → T-definable f → continuous f
corollary₀ f d = eloquent-is-continuous f (eloquence-theorem f d)

corollary₁ : ∀(f : Baire → ℕ) → T-definable f → uniformly-continuous(C-restriction f)
corollary₁ f d = eloquent-is-UC (C-restriction f) (eloquent-restriction f (eloquence-theorem f d))
```

This concludes the full, self-contained, MLTT proof in Agda notation, given from scratch. Because MLTT proofs are programs, we can run the two corollaries to compute moduli of (uniform) continuity of T-definable functions. Because MLTT itself has an interpretation in ZF(C), in which types are sets in the sense of classical mathematics, the results of this paper hold in classical mathematics too. Because the LaTeX source for this article [7] is simultaneously an Agda file that type-checks, the readers don't need to check the routine details of the proofs themselves, provided they trust the minimal core of Agda used here, and can instead concentrate on the interesting details of the constructions and proofs. One can envisage a future in which it will be easier to write (constructive and non-constructive) formal proofs than informal, rigorous proofs, letting our minds concentrate on the insights. This is certainly a provocative statement. But, in fact, the proof presented here was directly written in its formal form, without an informal draft other than a mental picture starting from the idea of generic sequence as described in the introduction, with some rudimentary help by Agda to perform the routine steps. Tactic-based systems such as e.g. Coq provide much more help, which in some instances can be considered as non-routine even if ultimately they are based on algorithms. But our principal motivation for writing this formal proof in an MLTT or realizability based computer system such as NuPrl, Coq, Lego, Agda, Minlog etc. is that mentioned above, that the proof is literally a program too, and hence can be used to compute moduli of (uniform) continuity, without the need to write a separate algorithm based on an informal, rigorous proof, as it is usually currently done, including by ourselves in previous work.

Having said that, it is useful to have a self-contained informal rigorous proof, which we include in the next section. Before that, we conclude this section by running our formal constructive proof for the purposes of illustration.

2.6 Experiments

To illustrate the concrete sense in which the above formal proof is constructive, we develop some experiments. These experiments are not meant to indicate the usefulness of the theorem proved above. They merely make clear that the theorems do have a concrete computational content.

First of all, given a term $t : (\iota \Rightarrow \iota) \Rightarrow \iota$, we can compute its modulus of (uniform) continuity.

```
mod-cont : T((ι ⇒ ι) ⇒ ι) → Baire → List ℕ
mod-cont t α = π₀(corollary₀ ⟦ t ⟧ (t , refl) α)
mod-cont-obs : ∀(t : T((ι ⇒ ι) ⇒ ι))(α : Baire) → mod-cont t α ≡ π₀(dialogue-continuity (dialogue-tree t) α)
mod-cont-obs t α = refl
```

```
infixl 0 _::_
infixl 1 _++_
_++_ : {X : Set} → List X → List X → List X
[] ++ u = u
(x :: t) ++ u = x :: t ++ u
flatten : {X : Set} → Tree X → List X
flatten empty = []
flatten (branch x t) = x :: flatten(t 0) ++ flatten(t 1)

mod-unif : T((ι ⇒ ι) ⇒ ι) → List ℕ
mod-unif t = flatten(π₀ (corollary₁ ⟦ t ⟧ (t , refl)))
```

The following Agda declaration allows us to write e.g. 3 rather than succ(succ(succ zero)):

```
{-# BUILTIN NATURAL ℕ #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC succ #-}
```

A difficulty we face is that it is not easy to write system T programs in the combinatory version of system T. Hence we start by developing some machinery.

```
I : ∀{σ : type} → T(σ ⇒ σ)
I {σ} = S · K · (K {σ} {σ})
I-behaviour : ∀{σ : type}{x : Set⟦ σ ⟧} → ⟦ I ⟧ x ≡ x
I-behaviour = refl

number : ℕ → T ι
number zero = Zero
number (succ n) = Succ · (number n)
```

Here is our first example:

```
t₀ : T((ι ⇒ ι) ⇒ ι)
t₀ = K · (number 17)
t₀-interpretation : ⟦ t₀ ⟧ ≡ λ α → 17
t₀-interpretation = refl
example₀ example₀' : List ℕ
example₀ = mod-cont t₀ (λ i → i)
example₀' = mod-unif t₀
```

These examples both evaluate to []. To provide more sophisticated examples, we work with an impoverished context ɣ that allows us to consider just one free variable v, which is represented by the I combinator:

```
v : ∀{ɣ : type} → T(ɣ ⇒ ɣ)
v = I
```

Application for such a context amounts to the S combinator:

```
infixl 1 _•_
_•_ : ∀{ɣ σ τ : type} → T(ɣ ⇒ σ ⇒ τ) → T(ɣ ⇒ σ) → T(ɣ ⇒ τ)
f • x = S · f · x

Number : ∀{ɣ} → ℕ → T(ɣ ⇒ ι)
Number n = K · (number n)
```

Here is an example:

```
t₁ : T((ι ⇒ ι) ⇒ ι)
t₁ = v • (Number 17)
t₁-interpretation : ⟦ t₁ ⟧ ≡ λ α → α 17
```

```
t₁-interpretation = refl
example₁ : List ℕ
example₁ = mod-unif t₁
```

This evaluates to 17 :: [].

```
t₂ : T((ι ⇒ ι) ⇒ ι)
t₂ = Rec • t₁ • t₁
t₂-interpretation : ⟦ t₂ ⟧ ≡ λ α → rec α (α 17) (α 17)
t₂-interpretation = refl
example₂ example₂' : List ℕ
example₂ = mod-unif t₂
example₂' = mod-cont t₂ (λ i → i)
```

These examples evaluate to 17 :: 17 :: 17 :: 0 :: 1 :: [] and to a list whose members are all 17.

```
Add : T(ι ⇒ ι ⇒ ι)
Add = Rec · Succ
infixl 0 _+_
_+_ : ∀{γ} → T(γ ⇒ ι) → T(γ ⇒ ι) → T(γ ⇒ ι)
x + y = K · Add • x • y

t₃ : T((ι ⇒ ι) ⇒ ι)
t₃ = Rec • (v • Number 1) • (v • Number 2 + v • Number 3)
t₃-interpretation : ⟦ t₃ ⟧ ≡ λ α → rec α (α 1) (rec succ (α 2) (α 3))
t₃-interpretation = refl
example₃ example₃' : List ℕ
example₃ = mod-cont t₃ succ
example₃' = mod-unif t₃
```

These examples evaluate to 3 :: 2 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: 8 :: [] and 3 :: 2 :: 1 :: 1 :: 0 :: 1 :: 2 :: 1 :: 0 :: 1 :: 1 :: 0 :: 0 :: 1 :: 1 :: 0 :: 1 :: [].

```
length : {X : Set} → List X → ℕ
length [] = 0
length (x :: s) = succ(length s)
max : ℕ → ℕ → ℕ
max 0 x = x
max x 0 = x
max (succ x) (succ y) = succ(max x y)
Max : List ℕ → ℕ
Max [] = 0
Max (x :: s) = max x (Max s)

t₄ : T((ι ⇒ ι) ⇒ ι)
t₄ = Rec • ((v • (v • Number 2)) + (v • Number 3)) • t₃
t₄-interpretation : ⟦ t₄ ⟧ ≡ λ α → rec α (rec succ (α (α 2)) (α 3)) (rec α (α 1) (rec succ (α 2) (α 3)))
t₄-interpretation = refl
example₄ example₄' : ℕ
example₄ = length(mod-unif t₄)
example₄' = Max(mod-unif t₄)
```

These examples evaluate to 215 and 3.

```
t₅ : T((ι ⇒ ι) ⇒ ι)
t₅ = Rec • (v • (v • t₂ + t₄)) • (v • Number 2)
t₅-explicitly : t₅ ≡ (S · (S · Rec · (S · I · (S · (S · (K · (Rec · Succ)) · (S · I · (S · (S · Rec ·
        (S · I · (K · (number 17))))· (S · I · (K · (number 17)))))) · (S · (S · Rec ·
        (S · (S · (K · (Rec · Succ)) · (S · I · (S · I · (K · (number 2))))) · (S · I ·
        (K · (number 3))))) · (S · (S · Rec · (S · I · (K · (number 1)))) · (S · (S ·
        (K · (Rec · Succ)) · (S · I · (K · (number 2)))) · (S · I · (K · (number 3)))))))) ·
        (S · I · (K · (number 2))))
t₅-explicitly = refl
t₅-interpretation : ⟦ t₅ ⟧ ≡ λ α → rec α (α(rec succ (α(rec α (α 17) (α 17))) (rec α (rec succ (α (α 2)) (α 3))
                                        (rec α (α 1) (rec succ (α 2) (α 3)))))) (α 2)
t₅-interpretation = refl
example₅ example₅' example₅'' : ℕ
example₅ = length(mod-unif t₅)
example₅' = Max(mod-unif t₅)
example₅'' = Max(mod-cont t₅ succ)
```

These examples evaluate to 15551, 17 and 57. All evaluations reported above are instantaneous, except this last set, which takes about a minute in a low-end netbook. The use of Church encoding of dialogue trees produces a dramatic performance improvement [7], with an instantaneous answer in these examples, because Klesli extension and the functor don't need to walk through trees to be performed.

3 Informal, rigorous proof

We now provide a self-contained, informal, rigorous version of the formal proof given above, in a foundationally neutral exposition.

We work with the combinatory version of (the term language of) Gödel's system T. We have a ground type ι and a right-associative type formation operation $- \Rightarrow -$. Every term as a unique type. We have the constants

(i) $\mathtt{Zero}\colon \iota$.

(ii) $\mathtt{Succ}\colon \iota \Rightarrow \iota$.

(iii) $\mathtt{Rec}_\sigma\colon (\sigma \Rightarrow \sigma) \Rightarrow \sigma \Rightarrow \iota \Rightarrow \sigma$.

(iv) $\mathtt{K}_{\sigma,\tau}\colon \sigma \Rightarrow \tau \Rightarrow \sigma$.

(v) $\mathtt{S}_{\rho,\sigma,\tau}\colon (\rho \Rightarrow \sigma \Rightarrow \tau) \Rightarrow (\rho \Rightarrow \sigma) \Rightarrow \rho \Rightarrow \tau$.

We omit the subscripts when they can be uniquely inferred from the context. If $t\colon \sigma \Rightarrow \tau$ and $u\colon \tau$ are terms, then so is $tu\colon \tau$, with the convention that this application operation is left associative. Write T_σ for the set of terms of type σ.

In the *standard interpretation*, we map a type expression σ to a set $[\![\sigma]\!]$ and a term $t\colon \sigma$ to an element $[\![t]\!] \in [\![\sigma]\!]$. These interpretations are defined by induction as follows:

$$[\![\iota]\!] = \mathbb{N}, \qquad [\![\sigma \Rightarrow \tau]\!] = [\![\tau]\!]^{[\![\sigma]\!]} = ([\![\sigma]\!] \to [\![\tau]\!]) \text{ (set of all functions } [\![\sigma]\!] \to [\![\tau]\!]),$$

$$[\![\mathtt{Zero}]\!] = 0, \qquad [\![\mathtt{Succ}]\!]n = n+1, \qquad [\![\mathtt{Rec}]\!]fxn = f^n(x),$$

$$[\![\mathtt{K}]\!]xy = x, \qquad [\![\mathtt{S}]\!]fgx = fx(gx), \qquad [\![tu]\!] = [\![t]\!]([\![u]\!]).$$

For any given three sets X, Y, Z, the set $\mathrm{D}\,XYZ$ of *dialogue trees* is inductively defined as follows:

(i) A leaf labeled by an element $z \in Z$ is a dialogue tree, written ηz.

(ii) If $\phi\colon Y \to \mathrm{D}\,XYZ$ is a Y-indexed family of dialogue trees and $x \in X$, then the tree with root labeled by x and with one branch leading to the subtree ϕy for each $y \in Y$ is also a dialogue tree, written $\beta\phi x$.

Such trees are well founded, meaning that every path from the root to a leaf is finite.

The above notation gives functions

$$\begin{aligned}&\eta : \ Z \to \mathrm{D}\, XYZ,\\ &\beta : \ (Y \to \mathrm{D}\, XYZ) \to X \to \mathrm{D}\, XYZ.\end{aligned}$$

Dialogue trees describe "computations" of functions $f \colon Y^X \to Z$. Leaves give answers, and labels of internal nodes are queries to an "oracle" $\alpha \in Y^X$, the argument of the function f. For any dialogue tree $d \in \mathrm{D}\, XYZ$, we inductively define a function $f_d \colon Y^X \to Z$ by

$$f_{\eta z}(\alpha) = z, \qquad f_{\beta\phi x}(\alpha) = f_{\phi(\alpha x)}(\alpha).$$

The functions $Y^X \to Z$ that arise in this way are called *eloquent*. Notice that the oracle α is queried finitely many times in this computation, because a dialogue tree is well founded. Hence the function $f = f_d \colon Y^X \to Z$ satisfies

$$\forall \alpha \in Y^X \quad \exists \text{finite } S \subseteq X \quad \forall \alpha' \in Y^X, \alpha =_S \alpha' \implies f\alpha = f\alpha',$$

where $\alpha =_S \alpha'$ is a shorthand for $\forall x \in S, \alpha x = \alpha' x$. When $X = Y = Z = \mathbb{N}$, this amounts to continuity in the product topology of $\mathbb{N}^{\mathbb{N}}$ with $\mathbb{N}$ discrete, which gives the Baire space.

For Y finite and X, Z arbitrary, the dialogue tree is finitely branching and hence finite by well-foundedness (or directly by induction), and so the set of potential queries to the oracle is finite, so that, for any $f = f_d \colon Y^X \to Z$ with Y finite,

$$\exists \text{finite } S \subseteq X \ \forall \alpha, \alpha' \in Y^X, \quad \alpha =_S \alpha' \implies f\alpha = f\alpha'.$$

When $Y = 2 = \{0, 1\}$ and $X = Z = \mathbb{N}$, this amounts to (uniform) continuity in the product topology of $2^{\mathbb{N}}$ with 2 discrete, which gives the Cantor space.

Clearly, any $\mathbb{N}$-branching tree $d \in \mathrm{D}\, \mathbb{N}\mathbb{N}\mathbb{N}$ can be pruned to a 2-branching tree $d' \in \mathrm{D}\, \mathbb{N}2\mathbb{N}$ so that $f_{d'} : 2^{\mathbb{N}} \to \mathbb{N}$ is the restriction of $f_d \colon \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ from sequences to binary sequences. Hence if we show that every T-definable function $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ is eloquent, we conclude that every T-definable function $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ is continuous and its restriction to $2^{\mathbb{N}}$ is uniformly continuous. For this purpose, we consider an auxiliary model of system T.

Define $\mathrm{B}\, X = \mathrm{D}\, \mathbb{N}\mathbb{N}X$. We remark that although B is the object part of a monad, as discussed in the introduction, it is not necessary to know this for the purposes of this proof. The data given below do obey the required laws to get a monad, but the details are left to the interested reader.

For any function $f \colon X \to \mathrm{B}\, Y$, inductively define $f^\sharp \colon \ \mathrm{B}\, X \to \mathrm{B}\, Y$ by

$$\begin{aligned}f^\sharp(\eta x) &= f x,\\ f^\sharp(\beta\phi i) &= \beta(\lambda j. f^\sharp(\phi j))i.\end{aligned}$$

This says that the tree $f^\sharp(d)$ is d with each leaf labeled by x replaced by the subtree fx, with no changes to the internal nodes. Given $f \colon X \to Y$, we define $f \colon \ \mathrm{B}\, X \to \mathrm{B}\, Y$ by

$$\mathrm{B}\, f = (\eta \circ f)^\sharp.$$

Hence $\mathrm{B}\, f(d)$ replaces each label x of a leaf of d by the label $f(x)$, and we have the naturality condition

$$\begin{tikzcd} \mathrm{B}\,X \arrow[r, "\mathrm{B}\,f"] & \mathrm{B}\,Y \\ X \arrow[u, "\eta"] \arrow[r, "f"'] & Y. \arrow[u, "\eta"'] \end{tikzcd}$$

For each $\alpha \in \mathbb{N}^{\mathbb{N}}$ and any set X, define a map $\mathrm{decode}_\alpha \colon \mathrm{B}\,X \to X$ by

$$\mathrm{decode}_\alpha(d) = f_d(\alpha).$$

Then, by definition, $\mathrm{decode}_\alpha(\eta x) = x$, and hence the naturality of η gives that of decode_α:

$$\begin{tikzcd} \mathrm{B}\,X \arrow[r, "\mathrm{B}\,f"] \arrow[d, "\mathrm{decode}_\alpha"'] & \mathrm{B}\,Y \arrow[d, "\mathrm{decode}_\alpha"] \\ X \arrow[r, "f"'] & Y. \end{tikzcd} \tag{1}$$

It is also easy to see, by induction on dialogue trees, that

$$\begin{tikzcd} \mathrm{B}\,X \arrow[rr, "f^\sharp"] \arrow[d, "\mathrm{decode}_\alpha"'] & & \mathrm{B}\,Y \arrow[d, "\mathrm{decode}_\alpha"] \\ X \arrow[r, "f"'] & \mathrm{B}\,Y \arrow[r, "\mathrm{decode}_\alpha"'] & Y. \end{tikzcd} \tag{2}$$

Now define

$$\begin{aligned} &\mathrm{generic} : \ \mathrm{B}\,\mathbb{N} \to \mathrm{B}\,\mathbb{N} \\ &\mathrm{generic} = (\beta\eta)^\sharp. \end{aligned}$$

Because $\beta \colon (\mathbb{N} \to \mathrm{B}\,\mathbb{N}) \to \mathbb{N} \to \mathrm{B}\,\mathbb{N}$ and $\eta \colon \mathbb{N} \to \mathrm{B}\,\mathbb{N}$, the function generic is well defined. Its crucial property is that

$$\begin{tikzcd} \mathrm{B}\,\mathbb{N} \arrow[r, "\mathrm{generic}"] \arrow[d, "\mathrm{decode}_\alpha"'] & \mathrm{B}\,\mathbb{N} \arrow[d, "\mathrm{decode}_\alpha"] \\ \mathbb{N} \arrow[r, "\alpha"'] & \mathbb{N}. \end{tikzcd} \tag{3}$$

The proof that

$$\mathrm{decode}_\alpha(\mathrm{generic}\, d) = \alpha(\mathrm{decode}_\alpha\, d)$$

is straightforward by induction on d.

Now define the B-interpretation of types as follows:

$$\mathrm{B}[\![\iota]\!] = \mathrm{B}([\![\iota]\!]) = \mathrm{B}\,\mathbb{N}, \qquad \mathrm{B}[\![\sigma \Rightarrow \tau]\!] = \mathrm{B}[\![\tau]\!]^{\mathrm{B}[\![\sigma]\!]}.$$

For any type σ and $f\colon X \to \mathrm{B}[\![\sigma]\!]$, define $f^\sharp\colon \mathrm{B}\,X \to \mathrm{B}[\![\sigma]\!]$ by induction on σ, where the base case $\sigma = \iota$ is given by the above definition, and the induction step $\sigma = (\rho \Rightarrow \tau)$ is given pointwise as

$$f^\sharp dy = (\lambda x. fxy)^\sharp d.$$

Notice that $f\colon X \to \mathrm{B}[\![\rho]\!] \to \mathrm{B}[\![\tau]\!]$ and $f^\sharp\colon\ \mathrm{B}\,X \to \mathrm{B}[\![\rho]\!] \to \mathrm{B}[\![\tau]\!]$.

Next extend system T with a new constant $\Omega\colon \iota \Rightarrow \iota$, a formal oracle, and define the B-interpretation of terms as follows:

$$\mathrm{B}[\![\Omega]\!] = \text{generic}, \quad \mathrm{B}[\![\mathtt{Zero}]\!] = \eta 0, \quad \mathrm{B}[\![\mathtt{Succ}]\!] = \mathrm{B}(\lambda n. n+1), \quad \mathrm{B}[\![\mathtt{Rec}]\!] fx = (\lambda n. f^n(x))^\sharp,$$

$$\mathrm{B}[\![\mathtt{K}]\!] xy = x, \qquad \mathrm{B}[\![\mathtt{S}]\!] fgx = fx(gx), \qquad \mathrm{B}[\![tu]\!] = \mathrm{B}[\![t]\!](\mathrm{B}[\![u]\!]).$$

We also need to consider the standard interpretation of system T extended with the oracle Ω. We treat the oracle as a free variable, as hence the value of this free variable has to be provided to define the interpretation:

$$[\![\Omega]\!]\alpha = \alpha, \qquad [\![\mathtt{Zero}]\!]\alpha = 0, \qquad [\![\mathtt{Succ}]\!]\alpha n = n+1, \qquad [\![\mathtt{Rec}]\!]\alpha fxn = f^n(x),$$

$$[\![\mathtt{K}]\!]\alpha xy = x, \qquad [\![\mathtt{S}]\!]\alpha fgx = fx(gx), \qquad [\![tu]\!]\alpha = [\![t]\!]\alpha([\![u]\!]\alpha).$$

We claim that for any term $t\colon \iota$,

$$[\![t]\!]\alpha = \mathrm{decode}_\alpha(\mathrm{B}[\![t]\!]).$$

To prove this, we work with a logical relation R_σ between functions $\mathbb{N}^\mathbb{N} \to [\![\sigma]\!]$ and elements of $\mathrm{B}[\![\sigma]\!]$ by induction on σ. For any $n\colon \mathbb{N}^\mathbb{N} \to \mathbb{N}$ and $n' \in \mathrm{B}\,\mathbb{N}$, we define

$$R_\iota nn' \iff \forall \alpha, n\alpha = \mathrm{decode}_\alpha\, n',$$

and, for any $f\colon \mathbb{N}^\mathbb{N} \to [\![\sigma]\!] \to [\![\tau]\!]$ and $f'\colon \mathrm{B}[\![\sigma]\!] \to \mathrm{B}[\![\tau]\!]$, we define

$$R_{\sigma\to\tau} ff' \iff \forall x\colon \mathbb{N}^\mathbb{N} \to [\![\sigma]\!],\ \forall x' : \mathrm{B}[\![\sigma]\!],\ R_\sigma xx' \to R_\tau(\lambda\alpha, f\alpha(x\alpha))(f'x').$$

We need a technical lemma for dealing with the dialogue interpretation of `Rec`:

Claim 3.1 *For all $g\colon \mathbb{N} \to \mathbb{N}^\mathbb{N} \to \mathrm{B}[\![\sigma]\!]$ and $g'\colon \mathbb{N} \to \mathrm{B}[\![\sigma]\!]$, if*

$$\forall k \in \mathbb{N},\ R_\sigma(gk)(g'k),$$

then $\forall n\colon \mathbb{N}^\mathbb{N} \to \mathbb{N},\ \forall n' \in \mathrm{B}\,\mathbb{N},\ R_\iota nn' \to R_\sigma(\lambda\alpha \to g(n\alpha)\alpha)(g'n')^\sharp$.

The proof is straightforward by induction on types, using diagram 2.

Claim 3.2 *$R_\sigma\ [\![t]\!]\ (\mathrm{B}[\![t]\!])$ for every term $t\colon \sigma$.*

The proof is by induction on terms, using diagram 3 for the term Ω, diagram 1 for the term `Succ`, and Claim 3.1 for the term `Rec`. The terms `K` and `S` are immediate but perhaps laborious, and the induction step, namely term application, is easy. This gives, in particular:

Claim 3.3 *For every term* $t\colon (\iota \Rightarrow \iota) \Rightarrow \iota$*, we have* $[\![t]\!]\alpha = \mathrm{decode}_\alpha(\textsc{b}[\![t\Omega]\!])$.

It follows that every T-definable function $f\colon \mathbb{N}^\mathbb{N} \to \mathbb{N}$ is eloquent, with dialogue tree given by $\textsc{b}[\![t\Omega]\!]$, where $t\colon (\iota \Rightarrow \iota) \Rightarrow \iota$ is any term denoting f, and hence continuous, with uniformly continuous restriction to $2^\mathbb{N}$.

4 Discussion, questions and conjectures

It may not be apparent from the informal proof of Section 3 that the argument is constructive, but Section 2 provides a constructive rendering in Martin-Löf type theory. We emphasize that our proof doesn't invoke the Fan Theorem [15,2] or any constructively contentious axiom.

We have deliberately chosen system T in its combinatory form as the simplest and most memorable non-trivial higher-type language to illustrate the essence of the technique proposed here. It is clearly routine (as well as interesting and useful) to apply the technique to a number of well-known extensions of the simply-typed lambda-calculus. But, for instance, at the time of writing, dependent types seem to require further thought, particularly in the presence of universes. Can one, e.g. (generalize and) apply the technique developed here to show that all MLTT definable functions $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ are continuous, and that their restrictions to $(\mathbb{N} \to 2)$ are uniformly continuous, in the main versions of (intensional) MLTT? More ambitiously, does the technique apply to Homotopy Type Theory [14]?

As pointed out by one of the anonymous referees, the syntactical techniques of [15] give more information: for any term t of type $(\iota \Rightarrow \iota) \Rightarrow \iota$ one can construct a term $m : (\iota \Rightarrow \iota) \Rightarrow \iota$ such that m internalizes the modulus of continuity of t. We adapted our technique to achieve this, as reported in [7], by working with Church encodings of dialogue trees defined within system T, and turning our semantical interpretation into a compositional translation of system T into itself. A corollary is that the dialogue trees of T-definable functions $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, being themselves T-definable, have height smaller than ϵ_0.

Acknowledgement

I benefitted from remarks on a previous version of this paper by Thierry Coquand, Dan Ghica, Achim Jung, Chuangjie Xu, and the anonymous referees.

References

[1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. Submitted for publication, 2012.

[2] M.J. Beeson. *Foundations of Constructive Mathematics*. Springer, 1985.

[3] E. Bishop. *Foundations of constructive analysis*. McGraw-Hill Book Co., New York, 1967.

[4] A. Bove and P. Dybjer. Dependent types at work. *Proceedings of Language Engineering and Rigorous Software Development, LNCS*, 5520:57–99, 2009.

[5] T. Coquand and G. Jaber. A note on forcing and type theory. *Fundam. Inf.*, 100(1-4):43–52, January 2010.

[6] T. Coquand and G. Jaber. A computational interpretation of forcing in type theory. In *Epistemology versus Ontology*, pages 203–213. Springer, 2012.

[7] M.H. Escardó. Continuity of Gödel's system T definable functionals via effectful forcing. Agda proof at http://www.cs.bham.ac.uk/~mhe/dialogue/, July 2012.

[8] P. Hancock, D. Pattinson, and N. Ghani. Representations of stream processors using nested fixed points. In *Logical Methods in Computer Science*, page 2009.

[9] W. A. Howard. Ordinal analysis of terms of finite type. *The Journal of Symbolic Logic*, 45:493–504, 1980.

[10] S.C. Kleene. Recursive functionals and quantifiers of finite types I. *Trans. Amer. Math. Soc*, 91, 1959.

[11] J. Longley. When is a functional program not a functional program? In *Proceedings of Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 1–7. ACM Press, 1999.

[12] P.Hancock and A. Setzer. Interactive programs in dependent type theory. In *CSL*, pages 317–331, 2000.

[13] G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11, 2003.

[14] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[15] A. S. Troelstra, editor. *Metamathematical investigation of intuitionistic arithmetic and analysis.* Lecture Notes in Mathematics, Vol. 344. Springer-Verlag, Berlin, 1973.

[16] C. Xu and M.H. Escardó. A constructive model of uniform continuity. To appear in TLCA, 2013.