Martin Escardo 2012-2013.

This is the file without comments for illustration. Maybe you wish to see

  http://www.cs.bham.ac.uk/~mhe/dialogue/dialogue.lagda
  http://www.cs.bham.ac.uk/~mhe/dialogue/dialogue.html
  http://www.cs.bham.ac.uk/~mhe/dialogue/dialogue.pdf

instead.

```
\begin{code}

module laconic-dialogue where

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)

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

data ℕ₂ : Set where
  ₀ : ℕ₂
  ₁ : ℕ₂

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)

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

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

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

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

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

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

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

Baire : Set
Baire = ℕ → ℕ

B : Set → Set
B = D ℕ ℕ
```

```
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'

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)

continuity-extensional : ∀(f g : Baire → ℕ) → (∀(α : Baire) → 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)

Cantor : Set
Cantor = ℕ → ℕ₂

C : Set → Set
C = D ℕ ℕ₂

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'

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)

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)))

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 _·_

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 ⟧

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

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

⟦_⟧' : ∀{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)

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)

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)

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

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

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

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

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

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

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))

zero' : B ℕ
zero' = η zero

succ' : B ℕ → B ℕ
succ' = B-functor succ

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)

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 ⟧

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

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)

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')

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

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 : Baire) → 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)

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

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))

\end{code}
```

This concludes the development. Some experiments follow (results not included, see the pdf version, or evaluate the examples please):

```
\begin{code}

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)))

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

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)

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

v : ∀{γ : type} → T(γ ⇒ γ)
v = I
```

```
infixl 1 _•_

_•_ : ∀{γ σ τ : type} → T(γ ⇒ σ ⇒ τ) → T(γ ⇒ σ) → T(γ ⇒ τ)
f • x = S · f · x

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

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

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)

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

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)

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)

\end{code}
```