# Strictly Monotone Brouwer Trees for Well Founded Recursion*

Anonymous Author(s)
Submission Id: icfp22main-p16-p

## Abstract

Gradual dependent types can help with the incremental adoption of dependently typed code by providing a principled semantics for *imprecise* types and proofs, where some parts have been omitted. Current theories of gradual dependent types, though, lack a central feature of type theory: propositional equality. Lennon-Bertrand et al. show that, when the reflexive proof *refl* is the only closed value of an equality type, a gradual extension of the Calculus of Inductive Constructions (CIC) with propositional equality violates static observational equivalences. Extensionally-equal functions should be indistinguishable at run time, but they can be distinguished using a combination of equality and type imprecision.

This work presents a gradual dependently typed language that supports propositional equality. We avoid the above issues by devising an equality type of which *refl* is not the only closed inhabitant. Instead, each equality proof is accompanied by a term that is at least as precise as the equated terms, acting as a witness of their plausible equality. These witnesses track partial type information as a program runs, raising errors when that information shows that two equated terms are undeniably inconsistent. Composition of type information is internalized as a construct of the language, and is deferred for function bodies whose evaluation is blocked by variables. We thus ensure that extensionally-equal functions compose without error, thereby preventing contexts from distinguishing them. We describe the challenges of designing consistency and precision relations for this system, along with solutions to these challenges. Finally, we prove important metatheory: type safety, conservative embedding of CIC, weak canonicity, and the gradual guarantees of Siek et al., which ensure that reducing a program's precision introduces no new static or dynamic errors.

*Title note


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/3547627

***Keywords*** dependent types, gradual types, propositional equality

### 0.0.1 Brouwer Trees

Unfortunately, it was not immediately apparent that any of the "off-the-shelf" formulations of constructive ordinals satisfied our critera, so we built our own formulation. We use a refined version of Brouwer trees:

```
data Ord  : Type where
    OZ : Ord
    O↑ : Ord → Ord
    OLim : (c : ℂ ℓ) → (El_Approx c → Ord) → Ord
```

There is a zero ordinal, a successor operator, and a limit ordinal that is the least upper bound of the image for a function from a code's type to ordinals. We borrow the trick of taking the limits over types (or in our case, codes) from **?**, since this lets us easily model the sizes of dependent functions and pairs. The ordering on these trees is defined following **?**:

```
data _ ≤o _ : Ord → Ord → Type where
    ≤oZ : (o : Ord) → OZ ≤o o
    ≤osucMono : (o1 : Ord) → (o2 : Ord) → o1 ≤o o2 → O↑ o1 ≤o O↑ o
    ≤ococone : (c : ℂ ℓ) → (o : Ord) → (f : El_Approx c → Ord) → (k : El_A
        → o ≤o f k → o ≤o OLim c f
    ≤olimiting : (o : Ord) → (c : ℂ ℓ) → (f : El_Approx c → Ord)
        → ((k : El_Approx c) → f k ≤o o) → OLim c f ≤o o


o1 <o o2 = O↑ o1 ≤o o2
```

That is, zero is the smallest ordinal, the successor is monotone, and the limit is actually the least upper bound of the function's image. Unlike **?**, we do not include transitivity as a rule, but we can prove it as a theorem. The maximum function on ordinals is defined as follows:

```
max_o : Ord → Ord → Ord
max_o OZ o = o
max_o o OZ = o
max_o (O↑ o1) (O↑ o2) = O↑ (max_o o1 o2)
max_o (OLim c f) o = OLim c (λk. max_o (f k) o)
max_o o (OLim c f) = OLim c (λk. max_o o (f k))
```

Long but straightforward proofs show that $max_o$ is monotone and computes and upper bound of its inputs. It reduces

when given O↑ for both inputs, so it is strictly monotone. However, we cannot prove that it is a least upper-bound. The problem is that limits are not well-behaved with respect to the maximum. We could instead construct the maximum using OLim, but this version would not be strictly monotone.

### 0.0.2 A Least Upper Bound

We solve the problems with $\max_o$ using a type of sizes, which include only the subset of ordinals that are idempotent with respect to the maximum. We can then define a type of sizes with the same interface as ordinals.

```
Size : Type
Size = (o : Ord) × (max_o o o ≤_o o)

_ ⋁ _ : Size → Size → Size
s₁ ⋁ s₂ = (max_o (fst s₁) (fst s₂), ...)

SZ : Size
SZ = (OZ, ≤_oZ)

S↑ : Size → Size
S↑ s = (O↑ (fst s), ≤_ssucMono (snd s))
```

Critically, the sizes are closed under the maximum operation: if $\max_o\ o_1\ o_1 \le_o\ o_1$ and $\max_o\ o_2\ o_2 \le_o\ o_2$, then $\max_o\ (\max_o\ o_1\ o_2)\ (\max_o\ o_1\ o_2) \le_o (\max_o\ o_1\ o_2)$. Zero and a successor operation for sizes are easily implemented. The difficulty is constructing a limit operator for sizes, since the self-idempotent ordinals are not closed under OLim. Our trick is to take the limit of maxing an ordinal with itself. We assume we have a code CℕN whose elements have an injection CtoℕN into ℕ. The natural numbers can be defined as an inductive type, but in our Agda development we add it as an extra code constructor. Having numbers lets us take the maximum of an ordinal with itself infinitely many times, resulting in an ordinal that is as large as the original but idempotent with respect to $\max_o$.

```
nmax : Ord → ℕ → Ord
nmax o Z  = OZ
nmax o (S n) = omax (nmax o n) o

max∞ : Ord → Ord
max∞ o = OLim CℕN (λk. nmax o (CtoℕN k))

max∞Idem : {o : Ord} → max_o (max∞ o) (max∞ o) ≤_o (max∞ o)

SLim : (c : ℂ ℓ) → (El_Approx c → Size) → Size
SLim c f = (max∞ (OLim c (λk. fst (f k))),  max∞Idem)
```

```
_ ≤_s _ : Size → Size → Size
s₁ ≤_s s₂ = (fst s₁) ≤_o (fst s₂)

_ <_s _ : Size → Size → Size
s₁ <_s s₂ = (S↑ s₁) ≤_s s₂

≤_strans : (s₁ : Size) → (s₂ : Size) → (s₃ : Size) →
      (s₁ ≤_s s₂) → (s₂ ≤_s s₃) → (s₁ ≤_s s₃)
≤_sZ : (s : Size) → SZ ≤_s s
≤_ssucMono : (s₁ : Size) → (s₂ : Size) → s₁ ≤_s s₂ → S↑ s₁ ≤_s S↑ s
≤_scocone : (c : ℂ ℓ) → (s : Size) → (f : El_Approx c → Size) → (k : El_Approx
      → s ≤_s f k → s ≤_s SLim c f
≤_slimiting : (s : Size) → (c : ℂ ℓ) → (f : El_Approx c → Size)
      → ((k : El_Approx c) → f k ≤_s s) → SLim c f ≤_s s

⋁ ≤: (s₁ : Size) → (s₂ : Size) → (s₁ ≤_s s₁ ⋁ s₂) × (s₂ ≤_2 s₁ ⋁ s₂)
⋁ mono : (s₁ : size) → (s₂ : Size) → (s₁' : Size) → (s₂' : Size)
      → (s₁ ≤_s s₁') → (s₂ ≤_s s₂') → (s₁ ⋁ s₂) ≤_s (s₁' ⋁ s₂')
⋁ idem : (s : Size) → (s ⋁ s) ≤_s s
⋁ lub : (s₁ : size) → (s₂ : size) → (s : Size)
      → (s₁ ≤_s s) → (s₂ ≤_s s) → (s₁ ⋁ s₂ ≤_s s)
```

**Figure 1.** Ordering on Sizes

Sizes satisfy all the same inequalities as raw ordinals, listed in Fig. 1. The monotonicity of ⋁ follows from the monotonicity of $\max_o$, and the idempotence of ⋁ follows by the definition of Size. Monotonicity, idempotence, and transitivity of $\le_s$ together imply that ⋁ is a least upper bound, and strict monotonicity follows from the strict monotonicity of $\max_o$.

## References