Strictly Monotone Brouwer Trees for Well Founded Recursion Over Multiple Values

Anonymous Author(s)

Abstract

Keywords: dependent types, gradual types, propositional equality

1 Introduction

1.1 Recursion and Dependent Types

Dependently typed languages, such as Agda [?], Coq [Bertot and Castéran 2004], Idris [?] and Lean [?], bridge the gap between theorem proving and programming.

Functions defined in dependently typed languages are typically required to be *total*: they must provably halt in all inputs. Since the halting problem is undecidable, recursively-defined functions must be written in such a way that the type checker can mechanically deduce termination. Some functions only make recursive calls to structurally-smaller arguments, so their termination is apparent to the compiler. However, some functions cannot be easily expressed using structural recursion. For such functions, the programmer must instead use *well founded recursion*, showing that there is some ordering, with no infinitely-descending chains, for which each recursive call is strictly smaller according to this ordering. For example, the typical quicksort algorithm is not structurally recursive, but can use well founded recursion on the length of the lists being sorted.

1.2 Ordinals

While numeric orderings work for first-order data, they cannot f

There are many formulations of ordinals in dependent type theory, each with their own advantages and disadvantages.

1.3 Contributions

This work defines *strictly monotone Brouwer Trees*, henceforth SMB-trees, a new presentation of ordinals that hit a sort of sweet-spot for defining functions by well founded recursion. Specifically, SMB-trees:

- are strictly ordered by a well founded relation;
- have a maximum operator which computes a least-upper bound;
- are *strictly-monotone* with respect to the maximum: if $a < b$ and $c < d$, then $\max\ a\ c < \max\ b\ d$;
- can compute the limits of arbitrary sequences;
- are light in axiomatic requirements: they are defined without using axiom K, univalence, quotient types, or higher inductive types.

1.4 Uses for SMB-trees

1.4.1 Well Founded Recursion. Having a maximum operator for ordinals is particularly useful when traversing over multiple higher order data structures in parallel, where neither argument takes priority over the other. In such a case, a lexicographic ordering cannot be used.

As an example, consider a unification algorithm over some encoding of types, and suppose that α-renaming or some other restriction prevents structural recursion from being used. To solve a unification problem $\Sigma(x : A).\,B = \Sigma(x : C).\,D$ we must recursively solve $A = C$ and $\forall x.\, B[x] = D[x]$. However, the type of x in the latter equation depends on the solution to the first equation, which is bounded by the size of the maximum of the sizes of both A and C. So for each recursive call to be on a smaller size, the size of $a = c$ and $b = d$ must both be strictly smaller than $(a, b) = (c, d)$. In a lexicographic ordering where the size of the left-hand size dominates, we know that a is strictly smaller than (a, b), but we have no guarantees that TODO. Conversely, if we order unification problems by the size of the maximum of their two sides.

This style of well founded induction was used to prove termination in a syntactic model of gradual dependent types [?]. There, Brouwer trees were used to establish termination of recursive procedures for combining the type information in two imprecise types. The decreasing metric was the maximum size of the codes for the types being combined. Brouwer trees' arbitrary limits were used to assign sizes to dependent function and product types, and the strict monotonicity of the maximum operator was essential for proving that recursive calls were on strictly smaller arguments.

1.4.2 Syntactic Models and Sized Types. An alternate way view of our contribution is as a tool for modelling sized types [?]. The implementation of sized types in Agda has been shown to be unsound [?], due to the interaction between propositional equality and the top size ∞ satisfying $\infty < \infty$. [Chan 2022] defines a dependently typed language with sized types that does not have a top size, proving it consistent using a syntactic model based on Brouwer trees.

SMB-trees provide the capability to extend existing syntactic models to sized types with a maximum operator. This brings the capability of consistent sized types closder to feature parity with Agda, which has a maximum operator for its sizes [?], while still maintaining logical consistency.

1.4.3 Algebraic Reasoning. Another advantage of SMB-trees is that they allow Brouwer trees to be interpreted using algebraic tools. SMB-trees can be described as In algebraic terminology, SMB-trees satisfy the following algebraic laws, up to the equivalence relation defined by $s \approx t := s \leq t \leq s$

- Join-semlattice: the binary max is associative, commutative, and idempotent
- Bounded: there is a least tree Z such that max $t\ Z \approx t$
- Inflationary endomorphism: there is a successor operator ↑ such that max $(\uparrow t)\ t \approx \uparrow t$ and $\uparrow(\max\ s\ t) = \max(\uparrow s)\ (\uparrow t)$

Bezem and Coquand [2022] describe a polynomial time algorithm for solving equations in such an algebra, and describe its usefulness for solving constraints involving universe levels in dependent type checking. While equations involving limits of infinite sequences are undecidable, the inflationary laws could be used to automatically discharge some equations involving sizes. This algebraic presentation is particularly amenable to solving equations using free extensions of algebras [Corbyn 2021; ?].

1.5 Implementation

We have implemented SMB-trees in Agda 2.6.4. Our library specifically avoids Agda-specific features such as cubcal type theory or Axiom K, so we expect that the library can be easily ported to other proof assistants.

This paper is written as a literate Agda document, and the definitions given in the paper are valid Agda code. Several definitions are presented with their body omitted due to space restrictions. The full implementation can be found in the supplementary materials section of this submission.

2 Brouwer Trees: An Introduction

Brouwer trees are a simple but elegant tool for proving termination of higher-order procedures. Traditionally, they are defined as follows:

```
data SmallTree : Set where
  Z : SmallTree
  ↑ : SmallTree → SmallTree
  Lim : (ℕ → SmallTree) → SmallTree
```

Under this definition, a Brouwer tree is either zero, the successor of another Brouwer tree, or the limit of a countable sequence of Brouwer trees. However, these are quite weak, in that they can only take the limit of countable sequences. To represent the limits of uncountable sequences, we can paramterize our definition over some Universe à la Tarski:

```
module RawTree {ℓ}
  (ℂ : Set ℓ)
  (El : ℂ → Set ℓ)
  (Cℕ : ℂ) (CℕIso : Iso (El Cℕ) ℕ ) where
```

Our module is paramterized over a universe level, a type ℂ of *codes*, and an "elements-of" interpretation function *El*, which computes the type represented by each code. We require that there be a code whose interpretation is isomorphic to the natural numbers, as this is essential to our construction in ??. Increasingly larger trees can be obtained by setting ℂ := Set ℓ and *El* := *id* for increasing ℓ. However, by defining an inductive-recursive universe, one can still capture limits over some non-countable types, since Tree is in Set whenever ℂ is.

We then generalize limits to any function whose domain is the interpretation of some code.

```
data Tree : Set ℓ where
  Z : Tree
  ↑ : Tree → Tree
  Lim : ∀ (c : ℂ ) → (f : El c → Tree) → Tree
```

The small limit constructor can be recovered from the natural-number code

```
ℕLim : (ℕ → Tree) → Tree
ℕLim f = Lim Cℕ (λ cn → f (Iso.fun CℕIso cn))
```

Brouwer trees are a the quintessential example of a higher-order inductive type.[1]: Each tree is built using smaller trees or functions producing smaller trees, which is essentially a way of storing a possibly infinite number of smaller trees.

2.1 Ordering Trees

Our ultimate goal is to have a well-founded ordering[2], so we define a relation to order Brouwer trees.

```
data _≤_ : Tree → Tree → Set ℓ where
  ≤-Z : ∀ {t} → Z ≤ t
  ≤-sucMono : ∀ {t1 t2}
    → t1 ≤ t2
    → ↑ t1 ≤ ↑ t2
  ≤-cocone : ∀ {t} {c : ℂ} (f : El c → Tree) (k : El c)
    → t ≤ f k
    → t ≤ Lim c f
  ≤-limiting : ∀ {t} {c : ℂ}
    → (f : El c → Tree)
    → (∀ k → f k ≤ t)
    → Lim c f ≤ t
```

This relation is reflexive:

```
≤-refl : ∀ t → t ≤ t
≤-refl Z = ≤-Z
≤-refl (↑ t) = ≤-sucMono (≤-refl t)
≤-refl (Lim c f)
  = ≤-limiting f (λ k → ≤-cocone f k (≤-refl (f k)))
```

[1]Not to be confused with Higher Inductive Types (HITs) from Homotopy Type Theory [Univalent Foundations Program 2013]

[2]Technically, this is a well-founded quasi-ordering because there are pairs of trees which are related by both ≤ and ≥, but which are not propositionally equal.

Crucially, it is also transitive, making the relation a preorder. We modify our the order relation from that of Kraus et al. [2023] so that transitivity can be proven constructively, rather than adding it as a constructor for the relation. This allows us to prove well-foundedness of the relation without needing quotient types or other advanced features.

```
≤-trans : ∀ {t1 t2 t3} → t1 ≤ t2 → t2 ≤ t3 → t1 ≤ t3
≤-trans ≤-Z p23 = ≤-Z
≤-trans (≤-sucMono p12) (≤-sucMono p23)
  = ≤-sucMono (≤-trans p12 p23)
≤-trans p12 (≤-cocone f k p23)
  = ≤-cocone f k (≤-trans p12 p23)
≤-trans (≤-limiting f x) p23
  = ≤-limiting f (λ k → ≤-trans (x k) p23)
≤-trans (≤-cocone f k p12) (≤-limiting .f x)
  = ≤-trans p12 (x k)
```

We create an infix version of transitivity for more readable construction of proofs:

```
_≤⨾_ : ∀ {t1 t2 t3} → t1 ≤ t2 → t2 ≤ t3 → t1 ≤ t3
lt1 ≤⨾ lt2 = ≤-trans lt1 lt2
```

2.1.1 Strict Ordering.

We can define a strictly-less-than relation in terms of our less-than relation and the successor constructor:

```
_<_ : Tree → Tree → Set ℓ
t1 < t2 = ↑ t1 ≤ t2
```

That is, a t_1 is strictly smaller than t_2 if the tree one-size larger than t_1 is as small as t_2. This relation has the properties one expects of a strictly-less-than relation: it is a transitive sub-relation of the less-than relation, every tree is strictly less than its successor, and no tree is strictly smaller than zero. JE ►TODO more?◄

```
≤↑t : ∀ t → t ≤ ↑ t
≤↑t Z = ≤-Z
≤↑t (↑ t) = ≤-sucMono (≤↑t t)
≤↑t (Lim c f)
  = ≤-limiting f λ k →
    (≤↑t (f k))
    ≤⨾ (≤-sucMono (≤-cocone f k (≤-refl (f k))))

<-in-≤ : ∀ {x y} → x < y → x ≤ y
<-in-≤ pf = ≤-trans (≤↑t _) pf

<∘≤-in-< : ∀ {x y z} → x < y → y ≤ z → x < z
<∘≤-in-< x<y y≤z = ≤-trans x<y y≤z

≤∘<-in-< : ∀ {x y z} → x ≤ y → y < z → x < z
≤∘<-in-< {x} {y} {z} x≤y y<z = ≤-trans (≤-sucMono x≤y) y<z

¬<Z : ∀ t → ¬(t < Z)
¬<Z t ()
```

2.2 Well Founded Induction

Recall the definition of a constructive well founded relation:

```
data Acc {A : Set a} (_<_ : A → A → Set ℓ) (x : A) : Set (a ⊠ ℓ) where
  acc : (rs : ∀ y → y < x → Acc _<_ y) → Acc _<_ x

WellFounded : (A → A → Set ℓ) → Set _
WellFounded _<_ = ∀ x → Acc _<_ x
```

That is, an element of a type is accessible for a relation if all strictly smaller elements of it are also accessible. A relation is well founded if all values are accessible with respect to that relation. This can then be used to define induction with arbitrary recursive calls on smaller values:

```
wfRec : (P : A → Set ℓ)
  → (∀ x → ((y : A) → y < x → P y) → P x)
  → ∀ x → P x
```

Following the construction of Kraus et al. [2023], we can show that the strict ordering on Brouwer trees is well founded. First, we prove a helper lemma: if a value is accessible, then all (not necessarily strictly) smaller terms are are also accessible.

```
smaller-accessible : (x : Tree)
  → Acc _<_ x → ∀ y → y ≤ x → Acc _<_ y
smaller-accessible x (acc r) y x<y
  = acc (λ y' y'<y → r y' (<∘≤-in-< y'<y x<y))
```

Then we use structural reduction to show that all terms are accesible. The key observations are that zero is trivially accessible, since no trees are strictly smaller than it, and that the only way to derive $\uparrow t \leq (\text{Lim } c\ f)$ is with ≤-cocone, yielding a concrete index k for which $\uparrow t \leq f\ k$, on which we can recur.

```
ordWF : WellFounded _<_
ordWF Z = acc λ _ ()
ordWF (↑ x)
  = acc (λ { y (≤-sucMono y≤x)
    → smaller-accessible x (ordWF x) y y≤x})
ordWF (Lim c f) = acc helper
  where
    helper : (y : Tree) → (y < Lim c f)
      → Acc _<_ y
    helper y (≤-cocone .f k y<fk)
      = smaller-accessible (f k)
        (ordWF (f k)) y (<-in-≤ y<fk)
```

3 First Attempts at a Join

In this section, we present two faulty implmentations of a join operator for trees. The first uses limits to define the join, but does not satisfy strict monotonicity. The second is defined inductively. Its satisfies strict monotonicity, but fails to be the least of all upper bounds, and requires us to assume that limits are only taken over non-empty types. In ??, we define SMB-trees a refinement of Brouwer trees that combines the benefits of both versions of the maximum.

3.1 Limit-based Maximum

Since the limit constructor finds the least upper bound of the image of a function, it should be possible to define the maximum of two trees as a special case of general limits. Indeed, we can compute the maximum of t_1 and t_2 as the limit of the function that produces t_1 when given 0 and t_2 otherwise.

```
limMax : Tree → Tree → Tree
limMax t1 t2 = ℕLim λ n → if0 n t1 t2
```

This version of the maximum has several of the properties we want from a maximum function: it is monotone, idempotent, commutative, and is a true least-upper-bound of its inputs.

```
limMax≤L : ∀ {t1 t2} → t1 ≤ limMax t1 t2
limMax≤L {t1} {t2}
  = ≤-cocone _ (Iso.inv CℕIso 0)
    (subst
      (λ x → t1 ≤ if0 x t1 t2)
      (sym (Iso.rightInv CℕIso 0))
      (≤-refl t1))


limMax≤R : ∀ {t1 t2} → t2 ≤ limMax t1 t2
limMax≤R {t1} {t2}
  = ≤-cocone _ (Iso.inv CℕIso 1)
    (subst
      (λ x → t2 ≤ if0 x t1 t2)
      (sym (Iso.rightInv CℕIso 1))
      (≤-refl t2))


limMaxIdem : ∀ {t} → limMax t t ≤ t
limMaxIdem {t} = ≤-limiting _ helper
  where
    helper : ∀ k → if0 (Iso.fun CℕIso k) t t ≤ t
    helper k with Iso.fun CℕIso k
    ... | zero = ≤-refl t
    ... | suc n = ≤-refl t

limMaxMono : ∀ {t1 t2 t1' t2'}
    → t1 ≤ t1' → t2 ≤ t2'
    → limMax t1 t2 ≤ limMax t1' t2'
limMaxMono {t1} {t2} {t1'} {t2'} lt1 lt2 = extLim _ _ helper
  where
    helper : ∀ k → if0 (Iso.fun CℕIso k) t1 t2 ≤ if0 (Iso.fun CℕIso k) t1' t2'
    helper k with Iso.fun CℕIso k
    ... | zero = lt1
    ... | suc n = lt2

limMaxLUB : ∀ {t1 t2 t} → t1 ≤ t → t2 ≤ t → limMax t1 t2 ≤ t
limMaxLUB lt1 lt2 = limMaxMono lt1 lt2 ≤⨾ limMaxIdem

limMaxCommut : ∀ {t1 t2} → limMax t1 t2 ≤ limMax t2 t1
limMaxCommut = limMaxLUB limMax≤R limMax≤L
```

3.1.1 Limitation: Strict Monotonicity.

The one crucial property that this formulation lacks is that it is not strictly monotone: we cannot deduce $\max\ t_1\ t_1 < \max\ t_1'\ t_2'$ from $t_1 < t_1'$ and $t_2 < t_2'$. This is because the only way to construct a proof that $\uparrow t \leq \mathsf{Lim}\ c\ f$ is using the ≤-cocone constructor. So we would need to prove that $\uparrow(\max\ t_1\ t_2) \leq t_1'$ or that $\uparrow(\max\ t_1\ t_2) \leq t_2'$, which cannot be deduced from the premises alone. What we want is to have $\uparrow\max\ (t_1)\ t_2 \leq \max(\uparrow t_1)\ (\uparrow t_2)$, so that strict monotonicity is a direct consequence of ordinary monotonicity of the maximum. This is not possible when defining the constructor as a limit.

3.2 Recursive Maximum

In our next attempt at defining a maximum operator, we obtain strict monotonicity by making $\max\ (\uparrow t_1)\ (\uparrow t_2) = \uparrow(\max\ t_1\ t_2)$ hold definitionally. Then, provided max is monotone, it will also be strictly monotone.

To do this, we compute the maximum of two trees recursively, pattern matching on the operands. We use a *view* [?] datatype to identify the cases we are matching on: we are matching on two arguments, which each have three possible constructors, but several cases overlap. Using a view type lets us avoid enumerating all nine possibilities when defining the maximum and proving its properties.

To begin, we parameterize our definition over a function yielding some element for any code's type.

```
module IndMax {ℓ}
  (ℂ : Set ℓ)
  (El : ℂ → Set ℓ)
  (Cℕ : ℂ) (CℕIso : Iso (El Cℕ) ℕ )
  (default : (c : ℂ) → El c) where
  open import RawTree ℂ El Cℕ CℕIso
```

We then define our view type:

```
private
  data IndMaxView : Tree → Tree → Set ℓ where
    IndMaxZ-L : ∀ {t} → IndMaxView Z t
    IndMaxZ-R : ∀ {t} → IndMaxView t Z
    IndMaxLim-L : ∀ {t} {c : ℂ} {f : El c → Tree}
      → IndMaxView (Lim c f) t
    IndMaxLim-R : ∀ {t} {c : ℂ} {f : El c → Tree}
      → (∀ {c' : ℂ} {f' : El c' → Tree} → ¬ (t ≡ Lim c' f'))
      → IndMaxView t (Lim c f)
    IndMaxLim-Suc : ∀ {t1 t2 } → IndMaxView (↑ t1) (↑ t2)
opaque
  indMaxView : ∀ t1 t2 → IndMaxView t1 t2
```

Our view type has five cases. The first two handle when either input is zero, and the second two handle when either input is a limit. The final case is when both inputs are successors. *indMaxView* computes the view for any pair of trees.

The maximum is then defined by pattern matching on the view for its arguments:

```
indMax : Tree → Tree → Tree
indMax' : ∀ {t1 t2} → IndMaxView t1 t2 → Tree

indMax t1 t2 = indMax' (indMaxView t1 t2)
indMax' {.Z} {t2} IndMaxZ-L = t2
indMax' {t1} {.Z} IndMaxZ-R = t1
indMax' {(Lim c f)} {t2} IndMaxLim-L
  = Lim c λ x → indMax (f x) t2
indMax' {t1} {(Lim c f)} (IndMaxLim-R _)
  = Lim c (λ x → indMax t1 (f x))
indMax' {(↑ t1)} {(↑ t2)} IndMaxLim-Suc = ↑ (indMax t1 t2)
```

The maximum of zero and *t* is always *t*, and the maximum of *t* and the limit of *f* is the limit of the function computing the maximum between *t* and *f x*. Finally, the maximum of two successors is the successor of the two maxima, giving the definitional equality we need for strict monotonicity.

This definition only works when limits of all codes are inhabited. The ≤-limiting constructor means that Lim *c f* ≤ Z whenever *El c* is uninhabited. So max ↑Z Lim *c f* will not actually be an upper bound for ↑Z if *c* has no inhabitants.

```
underLim : ∀ {c : ℂ} {t} → {f : El c → Tree} → (∀ k → t ≤ f k) → t ≤ Lim c f
  with () ← neq refl
underLim {c = c}   {t} {f} all = ≤-trans (≤-cocone (λ _ → t) (default c) (≤-refl t)) (≤-limiting (λ _ → t) (λ k → ≤-cocone f k (all k)))

opaque
  unfolding indMax indMax'

  indMax-≤L : ∀ {t1 t2} → t1 ≤ indMax t1 t2
  indMax-≤L {t1} {t2} with indMaxView t1 t2
  ... | IndMaxZ-L = ≤-Z
  ... | IndMaxZ-R = ≤-refl _
  ... | IndMaxLim-L {f = f} = extLim f (λ x → indMax (f x) t2) (λ k → indMax-≤L)
  ... | IndMaxLim-R {f = f} _ = underLim λ k → indMax-≤L {t2 = f k}
  ... | IndMaxLim-Suc = ≤-sucMono indMax-≤L

  indMax-≤R : ∀ {t1 t2} → t2 ≤ indMax t1 t2
  indMax-≤R {t1} {t2} with indMaxView t1 t2
  ... | IndMaxZ-R = ≤-Z
  ... | IndMaxZ-L = ≤-refl _
  ... | IndMaxLim-R {f = f} _ = extLim f (λ x → indMax t1 (f x)) (λ k → indMax-≤R {t1 = t1} {f k})
  ... | IndMaxLim-L {f = f} = underLim λ k → indMax-≤R
  ... | IndMaxLim-Suc {t1} {t2} = ≤-sucMono (indMax-≤R {t1 = t1} {t2 = t2})
```

```
indMax-monoR : ∀ {t1 t2 t2'} → t2 ≤ t2' → indMax t1 t2 ≤ indMax t1 t
indMax-monoR' : ∀ {t1 t2 t2'} → t2 < t2' → indMax t1 t2 < indMax (↑ t

indMax-monoR {t1} {t2} {t2'} lt with indMaxView t1 t2 in eq1 | indMaxV
... | IndMaxZ-L | v2 = ≤-trans lt (≤-reflEq (cong indMax' eq2))
... | IndMaxZ-R | v2 = ≤-trans indMax-≤L (≤-reflEq (cong indMax' eq2)
... | IndMaxLim-L {f = f1} | IndMaxLim-L = extLim _ _ λ k → indMax-r
indMax-monoR {t1} {(Lim _ f')} {.(Lim _ f)} (≤-cocone f k lt) | IndMaxL
  = ≤-limiting (λ x → indMax t1 (f' x)) (λ y → ≤-cocone (λ x → indM
indMax-monoR {t1} {.(Lim _ _)} {t2'} (≤-limiting f x₁) | IndMaxLim-R x
  ≤-trans (≤-limiting (λ x₂ → indMax t1 (f x₂)) λ k → indMax-monoR
indMax-monoR {(↑ t1)} {.(↑ _)} {.(↑ _)} (≤-sucMono lt) | IndMaxLim-Suc
indMax-monoR {(↑ t1)} {(↑ t2)} {(Lim _ f)} (≤-cocone f k lt) | IndMaxLin
  = ≤-trans (indMax-monoR' {t1 = t1} {t2 = t2} {t2' = f k} lt) (≤-cocone (

indMax-monoR' {t1} {t2} {t2'} (≤-sucMono lt) = ≤-sucMono ( (indMax-r
indMax-monoR' {t1} {t2} {.(Lim _ f)} (≤-cocone f k lt)
  = ≤-cocone _ k (indMax-monoR' {t1 = t1} lt)


indMax-monoL : ∀ {t1 t1' t2} → t1 ≤ t1' → indMax t1 t2 ≤ indMax t1' t
indMax-monoL' : ∀ {t1 t1' t2} → t1 < t1' → indMax t1 t2 < indMax t1'
indMax-monoL {t1} {t1'} {t2} lt with indMaxView t1 t2 in eq1 | indMaxV
... | IndMaxZ-L | v2 = ≤-trans (indMax-≤R {t1 = t1'}) (≤-reflEq (cong ind
... | IndMaxZ-R | v2 = ≤-trans lt (≤-trans (indMax-≤L {t1 = t1'}) (≤-reflE
indMax-monoL {.(Lim _ _)} {.(Lim _ f)} {t2} (≤-cocone f k lt) | IndMaxL
  = ≤-cocone (λ x → indMax (f x) t2) k (indMax-monoL lt)
indMax-monoL {.(Lim _ _)} {t1'} {t2} (≤-limiting f lt) | IndMaxLim-L |
  = ≤-limiting (λ x₁ → indMax (f x₁) t2) λ k → ≤-trans (indMax-mono
indMax-monoL {.Z} {.Z} {.(Lim _ _)} ≤-Z | IndMaxLim-R neq | IndMaxZ
indMax-monoL {.(Lim _ f)} {.Z} {.(Lim _ _)} (≤-limiting f x) | IndMaxLin

indMax-monoL {t1} {.(Lim _ _)} {.(Lim _ _)} (≤-cocone _ k lt) | IndMaxL
  = ≤-limiting (λ x → indMax t1 (f x)) (λ y → ≤-cocone (λ x → indMa
  (≤-trans (indMax-monoL lt) (indMax-monoR {t1 = f' k} (≤-cocone f y
... | IndMaxLim-R neq | IndMaxLim-R {f = f} neq' = extLim (λ x → ind
indMax-monoL {.(↑ _)} {.(↑ _)} {.(↑ _)} (≤-sucMono lt) | IndMaxLim-Suc
  = ≤-sucMono (indMax-monoL lt)
indMax-monoL {.(↑ _)} {.(Lim _ f)} {.(↑ _)} (≤-cocone f k lt) | IndMaxLin
  = ≤-cocone (λ x → indMax (f x) (↑ _)) k (indMax-monoL' lt)

indMax-monoL' {t1} {t1'} {t2} lt with indMaxView t1 t2 in eq1 | indMax
indMax-monoL' {t1} {.(↑ _)} {t2} (≤-sucMono lt) | v1 | v2 = ≤-sucMono (≤
indMax-monoL' {t1} {.(Lim _ f)} {t2} (≤-cocone f k lt) | v1 | v2
  = ≤-cocone _ k (≤-trans (≤-sucMono (≤-reflEq (cong indMax' (sym e
```

3.2.1 Limitation: Idempotence.

```
indMax-mono : ∀ {t1 t2 t1' t2'} → t1 ≤ t1' → t2 ≤ t2' → indMax t1 t2 ≤ i
indMax-mono {t1' = t1'} lt1 lt2 = ≤-trans (indMax-monoL lt1) (indMax-m

indMax-strictMono : ∀ {t1 t2 t1' t2'} → t1 < t1' → t2 < t2' → indMax t1
```

```
indMax-strictMono lt1 lt2 = indMax-mono lt1 lt2

indMax-sucMono : ∀ {t1 t2 t1' t2'} → indMax t1 t2 ≤ indMax t1' t2' → indMax t1 t2 < indMax (↑ t1') (↑ t2')
indMax-sucMono lt = ≤-sucMono lt

indMax-Z : ∀ t → indMax t Z ≤ t
indMax-Z Z = ≤-Z
indMax-Z (↑ t) = ≤-refl (indMax (↑ t) Z)
indMax-Z (Lim c f) = extLim (λ x → indMax (f x) Z) f (λ k → indMax-Z (f k))

indMax-↑ : ∀ {t1 t2} → indMax (↑ t1) (↑ t2) ≡ ↑ (indMax t1 t2)
indMax-↑ = refl

indMax-≤Z : ∀ t → indMax t Z ≤ t
indMax-≤Z Z = ≤-refl _
indMax-≤Z (↑ t) = ≤-refl _
indMax-≤Z (Lim c f) = extLim _ _ (λ k → indMax-≤Z (f k))


indMax-limR : ∀ {c : ℂ} (f : El c → Tree) t → indMax t (Lim c f) ≤ Lim c (λ k → indMax t (f k))
indMax-limR f Z = ≤-refl _
indMax-limR f (↑ t) = extLim _ _ λ k → ≤-refl _
indMax-limR f (Lim c f₁) = ≤-limiting _ λ k → ≤-trans (indMax-limR f (f₁ k)) (extLim _ _ (λ k2 → indMax-monoL {t1 = f₁ k} {t1' = Lim c f

indMax-commut : ∀ t1 t2 → indMax t1 t2 ≤ indMax t2 t1
indMax-commut t1 t2 with indMaxView t1 t2
... | IndMaxZ-L = indMax-≤L
... | IndMaxZ-R = ≤-refl _
... | IndMaxLim-R {f = f} x = extLim _ _ (λ k → indMax-commut t1 (f k))
... | IndMaxLim-Suc {t1 = t1} {t2 = t2} = ≤-sucMono (indMax-commut t1 t2)
... | IndMaxLim-L {c = c} {f = f} with indMaxView t2 t1
... | IndMaxZ-L = extLim _ _ λ k → indMax-Z (f k)
... | IndMaxLim-R x = extLim _ _ (λ k → indMax-commut (f k) t2)
... | IndMaxLim-L {c = c2} {f = f2} =
  ≤-trans (extLim _ _ λ k → indMax-limR f2 (f k))
  (≤-trans (≤-limiting _ (λ k → ≤-limiting _ λ k2 → ≤-cocone _ k2 (≤-cocone _ k (≤-refl _))))
  (≤-trans (≤-refl (Lim c2 λ k2 → Lim c λ k → indMax (f k) (f2 k2)))
  (extLim _ _ (λ k2 → ≤-limiting _ λ k1 → ≤-trans (indMax-commut (f k1) (f2 k2)) (indMax-monoR {t1 = f2 k2} {t2 = f k1} {t2' = Lim c f}

indMax-assocL : ∀ t1 t2 t3 → indMax t1 (indMax t2 t3) ≤ indMax (indMax t1 t2) t3
indMax-assocL t1 t2 t3 with indMaxView t2 t3 in eq23
... | IndMaxZ-L = indMax-monoL {t1 = t1} {t1' = indMax t1 Z} {t2 = t3} indMax-≤L
... | IndMaxZ-R = indMax-≤L
... | m with indMaxView t1 t2
... | IndMaxZ-L rewrite sym eq23 = ≤-refl _
... | IndMaxZ-R rewrite sym eq23 = ≤-refl _
... | IndMaxLim-R {f = f} x rewrite sym eq23 = ≤-trans (indMax-limR (λ x → indMax (f x) t3) t1) (extLim _ _ λ k → indMax-assocL t1 (f
indMax-assocL .(↑ _) .(↑ _) .Z | IndMaxZ-R | IndMaxLim-Suc = ≤-refl _
indMax-assocL t1 t2 .(Lim _ _) | IndMaxLim-R {f = f} x | IndMaxLim-Suc = extLim _ _ λ k → indMax-assocL t1 t2 (f k)
indMax-assocL (↑ t1) (↑ t2) (↑ t3) | IndMaxLim-Suc | IndMaxLim-Suc = ≤-sucMono (indMax-assocL t1 t2 t3)
... | IndMaxLim-L {f = f} rewrite sym eq23 = extLim _ _ λ k → indMax-assocL (f k) t2 t3
```

```
indMax-assocR : ∀ t1 t2 t3 → indMax (indMax t1 t2) t3 ≤ indMax t1 (ind
indMax-assocR t1 t2 t3 = ≤-trans (indMax-commut (indMax t1 t2) t3) (≤-
  (≤-trans (indMax-assocL t3 t2 t1) (≤-trans (indMax-commut (indMax 

indMax-swap4 : ∀ {t1 t1' t2 t2'} → indMax (indMax t1 t1') (indMax t2 t2
indMax-swap4 {t1}{t1'}{t2 }{t2'} =
  indMax-assocL (indMax t1 t1') t2 t2'
  ≤⨾ indMax-monoL {t1 = indMax (indMax t1 t1') t2} {t2 = t2'}
  (indMax-assocR t1 t1' t2 ≤⨾ indMax-monoR {t1 = t1} (indMax-commu
  ≤⨾ indMax-assocR (indMax t1 t2) t1' t2'

indMax-swap6 : ∀ {t1 t2 t3 t1' t2' t3'} → indMax (indMax t1 t1') (indMax
indMax-swap6 {t1} {t2} {t3} {t1'} {t2'} {t3'} =
  indMax-monoR {t1 = indMax t1 t1'} (indMax-swap4 {t1 = t2} {t1' = t2'}
  ≤⨾ indMax-swap4 {t1 = t1} {t1' = t1'}

indMax-lim2L :
  ∀
  {c1 : ℂ}
  (f1 : El c1 → Tree)
  {c2 : ℂ}
  (f2 : El c2 → Tree)
  → Lim c1 (λ k1 → Lim c2 (λ k2 → indMax (f1 k1) (f2 k2))) ≤ indMax (
indMax-lim2L f1 f2 = ≤-limiting _ (λ k1 → ≤-limiting _ λ k2 → indMax-

indMax-lim2R :
  ∀
  {c1 : ℂ}
  (f1 : El c1 → Tree)
  {c2 : ℂ}
  (f2 : El c2 → Tree)
  → indMax (Lim c1 f1) (Lim    c2 f2) ≤ Lim c1 (λ k1 → Lim c2 (λ k2 →
indMax-lim2R f1 f2 = extLim _ _ (λ k1 → indMax-limR _ (f1 k1))
```

4 Trees with a Striclty-Monotone Idempotent Join

4.1 Well-Behaved Trees

```
opaque
  unfolding indMax indMax'
  --Attempt to have an idempotent version of indMax

  nindMax : Tree → ℕ → Tree
  nindMax t ℕ.zero = Z
  nindMax t (ℕ.suc n) = indMax (nindMax t n) t

  nindMax-mono : ∀ {t1 t2 } n → t1 ≤ t2 → nindMax t1 n ≤ nindMax t2
  nindMax-mono ℕ.zero lt = ≤-Z
  nindMax-mono {t1 = t1} {t2} (ℕ.suc n) lt = indMax-mono {t1 = nindMa
```

```
--
indMax∞ : Tree → Tree
indMax∞ t = ℕLim (λ n → nindMax t n)

indMax-∞lt1 : ∀ t → indMax (indMax∞ t) t ≤ indMax∞ t
indMax-∞lt1 t = ≤-limiting _ λ k → helper (Iso.fun CℕIso k)
  where
  helper : ∀ n → indMax (nindMax t n) t ≤ indMax∞ t
  helper n = ≤-cocone _ (Iso.inv CℕIso (ℕ.suc n)) (subst (λ sn → nindMax t (ℕ.suc n) ≤ nindMax t sn) (sym (Iso.rightInv CℕIso (suc n)))
indMax-∞ltn : ∀ n t → indMax (indMax∞ t) (nindMax t n) ≤ indMax∞ t
indMax-∞ltn ℕ.zero t = indMax-≤Z (indMax∞ t)
indMax-∞ltn (ℕ.suc n) t =
  ≤-trans (indMax-monoR {t1 = indMax∞ t} (indMax-commut (nindMax t n) t))
  (≤-trans (indMax-assocL (indMax∞ t) t (nindMax t n))
  (≤-trans (indMax-monoL {t1 = indMax (indMax∞ t) t} {t2 = nindMax t n} (indMax-∞lt1 t)) (indMax-∞ltn n t)))
indMax∞-idem : ∀ t → indMax (indMax∞ t) (indMax∞ t) ≤ indMax∞ t
indMax∞-idem t = ≤-limiting _ λ k → ≤-trans (indMax-commut (nindMax t (Iso.fun CℕIso k)) (indMax∞ t)) (indMax-∞ltn (Iso.fun Cℕ

indMax∞-self : ∀ t → t ≤ indMax∞ t
indMax∞-self t = ≤-cocone _ (Iso.inv CℕIso 1) (subst (λ x → t ≤ nindMax t x) (sym (Iso.rightInv CℕIso 1)) (≤-refl _))

indMax∞-idem∞ : ∀ t → indMax t t ≤ indMax∞ t
indMax∞-idem∞ t = ≤-trans (indMax-mono (indMax∞-self t) (indMax∞-self t)) (indMax∞-idem t)

indMax∞-mono : ∀ {t1 t2} → t1 ≤ t2 → (indMax∞ t1) ≤ (indMax∞ t2)
indMax∞-mono lt = extLim _ _ λ k → nindMax-mono (Iso.fun CℕIso k) lt


nindMax-≤ : ∀ {t} n → indMax t t ≤ t → nindMax t n ≤ t
nindMax-≤ ℕ.zero lt = ≤-Z
nindMax-≤ {t = t} (ℕ.suc n) lt = ≤-trans (indMax-monoL {t1 = nindMax t n} {t2 = t} (nindMax-≤ n lt)) lt

indMax∞-≤ : ∀ {t} → indMax t t ≤ t → indMax∞ t ≤ t
indMax∞-≤ lt = ≤-limiting  _ λ k → nindMax-≤ (Iso.fun CℕIso k) lt

-- Convenient helper for turning < with indMax□ into < without
indMax<-∞ : ∀ {t1 t2 t} → indMax (indMax∞ (t1)) (indMax∞ t2) ≤ t → indMax t1 t2 < t
indMax<-∞ lt = ≤∘<-in-< (indMax-mono (indMax∞-self _) (indMax∞-self _)) lt

indMax-<Ls : ∀ {t1 t2 t1' t2'} → indMax t1 t2 < indMax (↑ (indMax t1 t1')) (↑ (indMax t2 t2'))
indMax-<Ls {t1} {t2} {t1'} {t2'} = indMax-sucMono {t1 = t1} {t2 = t2} {t1' = indMax t1 t1'} {t2' = indMax t2 t2'}
  (indMax-mono {t1 = t1} {t2 = t2} (indMax-≤L) (indMax-≤L))

indMax∞-<Ls : ∀ {t1 t2 t1' t2'} → indMax t1 t2 < indMax (↑ (indMax (indMax∞ t1) t1')) (↑ (indMax (indMax∞ t2) t2'))
indMax∞-<Ls {t1} {t2} {t1'} {t2'} = <∘≤-in-< (indMax-<Ls {t1} {t2} {t1'} {t2'})
  (indMax-mono {t1 = ↑ (indMax t1 t1')} {t2 = ↑ (indMax t2 t2')}
  (≤-sucMono (indMax-monoL (indMax∞-self t1)))
  (≤-sucMono (indMax-monoL (indMax∞-self t2))))

indMax∞-lub : ∀ {t1 t2 t} → t1 ≤ indMax∞ t → t2 ≤ indMax∞ t → indMax t1 t2 ≤ (indMax∞ t)
indMax∞-lub {t1 = t1} {t2 = t2} lt1 lt2 = indMax-mono {t1 = t1} {t2 = t2} lt1 lt2 ≤⨾ indMax∞-idem _

indMax∞-absorbL : ∀ {t1 t2 t} → t2 ≤ t1 → t1 ≤ indMax∞ t → indMax t1 t2 ≤ indMax∞ t
indMax∞-absorbL lt12 lt1 = indMax∞-lub lt1 (lt12 ≤⨾ lt1)
```

```
indMax∞-distL : ∀ {t1 t2} → indMax (indMax∞ t1) (indMax∞ t2) ≤ ind
indMax∞-distL {t1} {t2} =
  indMax∞-lub {t1 = indMax∞ t1} {t2 = indMax∞ t2} (indMax∞-mono

indMax∞-distR : ∀ {t1 t2} → indMax∞ (indMax t1 t2) ≤ indMax (indM
indMax∞-distR {t1} {t2} = ≤-limiting _ λ k → helper {n = Iso.fun CℕIso
  where
  helper : ∀ {t1 t2 n} → nindMax (indMax t1 t2) n ≤ indMax (indMax∞
  helper {t1} {t2} {ℕ.zero} = ≤-Z
  helper {t1} {t2} {ℕ.suc n} =
    indMax-monoL {t1 = nindMax (indMax t1 t2) n} (helper {t1 = t1} {t2
    ≤⨾ indMax-swap4 {indMax∞ t1} {indMax∞ t2} {t1} {t2}
    ≤⨾ indMax-mono {t1 = indMax (indMax∞ t1) t1} {t2 = indMax (ind
    (indMax∞-lub {t1 = indMax∞ t1} (≤-refl _) (indMax∞-self _))
    (indMax∞-lub {t1 = indMax∞ t2} (≤-refl _) (indMax∞-self _)))

indMax∞-cocone : ∀ {c : ℂ} (f : El c → Tree) k →
  f k ≤ indMax∞ (Lim c f)
indMax∞-cocone f k = indMax∞-self _ ≤⨾ indMax∞-mono (≤-cocone
```

5 A Strictly-Monotone, Idempotent Join

```
module Idem {ℓ}
  (ℂ : Set ℓ)
  (El : ℂ → Set ℓ)
  (Cℕ : ℂ) (CℕIso : Iso (El Cℕ) ℕ ) where

module Raw where
  open import RawTree ℂ (λ c → Maybe (El c)) Cℕ (maybeNatIso CℕIs


record Tree : Set ℓ where
  constructor MkTree
  field
    sTree : Raw.Tree
    sIdem : (indMax sTree sTree) Raw.≤ sTree
open Tree

record _≤_ (s1 s2 : Tree) : Set ℓ where
  constructor mk≤
  inductive
  field
    get≤ : (sTree s1) Raw.≤ (sTree s2)
open _≤_

opaque
  unfolding indMax

  Z : Tree
  Z = MkTree Raw.Z Raw.≤-Z
  ↑ : Tree → Tree
  ↑ (MkTree o pf) = MkTree (Raw.↑ o) ( subst (λ x → x Raw.≤ Raw.↑ o) (sym indMax-↑) (Raw.≤-sucMono pf) )

  ≤↑ : ∀ s → s ≤ ↑ s
  ≤↑ s = mk≤ (Raw.≤↑t _)

_<_ : Tree → Tree → Set ℓ
_<_ s1 s2 = (↑ s1) ≤ s2

opaque
  unfolding indMax Z ↑ indMaxView
  max : Tree → Tree → Tree
  max s1 s2 = MkTree (indMax (sTree s1) (sTree s2)) (indMax-swap4 Raw.≤ ⨾ indMax-mono (sIdem s1) (sIdem s2))

  Lim : ∀     (c : ℂ) → (f : El c → Tree) → Tree
  Lim c f =
    MkTree
    (indMax∞ (Raw.Lim c (maybe′ (λ x → sTree (f x)) Raw.Z)))
    (indMax∞-idem _)

--MkTree (indMax□ (Lim c (□ x → sTree (f x)))) ( indMax□-idem (Lim c (□ x → sTree (f x))) )

  ≤-Z : ∀ {s} → Z ≤ s
  ≤-Z = mk≤ Raw.≤-Z

  ≤-sucMono : ∀ {s1 s2} → s1 ≤ s2 → ↑ s1 ≤ ↑ s2
  ≤-sucMono (mk≤ lt) = mk≤ (Raw.≤-sucMono lt)

  infixr 10 _≤⨾_
  _≤⨾_ : ∀ {s1 s2 s3} → s1 ≤ s2 → s2 ≤ s3 → s1 ≤ s3
  _≤⨾_ (mk≤ lt1) (mk≤ lt2) = mk≤ (Raw.≤-trans lt1 lt2)

  ≤-refl : ∀ {s} → s ≤ s
  ≤-refl =    mk≤ (Raw.≤-refl _)

  ≤-limUpperBound : ∀ {c : ℂ} → {f : El c → Tree}
    → ∀ k → f k ≤ Lim c f
  ≤-limUpperBound {c = c} {f = f} k = mk≤ (Raw.≤-cocone _ (just k) (Raw.≤-refl _) Raw.≤⨾ indMax∞-self (Raw.Lim c _))

  ≤-limLeast : ∀ {c : ℂ} → {f : El c → Tree}
    → {s : Tree}
    → (∀ k → f k ≤ s) → Lim c f ≤ s
  ≤-limLeast {f = f} {s = MkTree o idem} lt
    = mk≤ (
        indMax∞-mono (Raw.≤-limiting _ (maybe (λ k → get≤ (lt k)) Raw.≤-Z))
        Raw.≤⨾ (indMax∞-≤ idem) )

  ≤-extLim : ∀ {c : ℂ} → {f1 f2 : El c → Tree}
    → (∀ k → f1 k ≤ f2 k)
    → Lim c f1 ≤ Lim c f2
  ≤-extLim lt = ≤-limLeast (λ k → lt k ≤⨾ ≤-limUpperBound k)

  ≤-extExists : ∀ {c1 c2 : ℂ} → {f1 : El c1 → Tree} {f2 : El c2 → Tree}
    → (∀ k1 → Σ[ k2 ∈ El c2 ] f1 k1 ≤ f2 k2)
    → Lim c1 f1 ≤ Lim c2 f2
  ≤-extExists {f1 = f1} {f2} lt = ≤-limLeast (λ k1 → proj₂ (lt k1) ≤⨾ ≤-limU

  --□-limLeast (□ k1 → proj□ (lt k1) □ ⨾ □-limUpperBound (pr
  ¬Z<↑ : ∀    s → ¬ ((↑ s) ≤ Z)
  ¬Z<↑ s pf = Raw.¬<Z (sTree s) (get≤ pf)

  max-≤L : ∀ {s1 s2} → s1 ≤ max s1 s2
  max-≤L = mk≤ indMax-≤L

  max-≤R : ∀ {s1 s2} → s2 ≤ max s1 s2
  max-≤R = mk≤ indMax-≤R

  max-mono : ∀ {s1 s1’ s2 s2’} → s1 ≤ s1’ → s2 ≤ s2’ →
    max s1 s2 ≤ max s1’ s2’
  max-mono lt1 lt2 = mk≤ (indMax-mono (get≤ lt1) (get≤ lt2))

  max-monoR : ∀ {s1 s2 s2’} → s2 ≤ s2’ → max s1 s2 ≤ max s1 s2’
  max-monoR {s1} {s2} {s2’} lt = max-mono {s1 = s1} {s1’ = s1} {s2 = s2} {s2

  max-monoL : ∀ {s1 s1’ s2} → s1 ≤ s1’ → max s1 s2 ≤ max s1’ s2
  max-monoL {s1} {s1’} {s2} lt = max-mono {s1} {s1’} {s2} {s2} lt (≤-refl {s2}

  max-idem : ∀ {s} → max s s ≤ s
  max-idem {s = MkTree o pf} = mk≤ pf

  max-LUB : ∀ {t1 t2 t} → t1 ≤ t → t2 ≤ t → max t1 t2 ≤ t
  max-LUB lt1 lt2 = max-mono lt1 lt2 ≤⨾ max-idem

  max-commut : ∀ s1 s2 → max s1 s2 ≤ max s2 s1
  max-commut s1 s2 = mk≤ (indMax-commut (sTree s1) (sTree s2))

  max-assocL : ∀ s1 s2 s3 → max s1 (max s2 s3) ≤ max (max s1 s2) s3
  max-assocL s1 s2 s3 = mk≤ (indMax-assocL _ _ _)

  max-assocR : ∀ s1 s2 s3 → max (max s1 s2) s3 ≤ max s1 (max s2 s3)
  max-assocR s1 s2 s3 = mk≤ (indMax-assocR _ _ _)

  max-swap4 : ∀ {s1 s1’ s2 s2’} → max (max s1 s1’) (max s2 s2’) ≤ max (m
  max-swap4 = mk≤ indMax-swap4

  max-strictMono : ∀ {s1 s1’ s2 s2’} → s1 < s1’ → s2 < s2’ → max s1 s2 <
  max-strictMono lt1 lt2 = mk≤ (indMax-strictMono (get≤ lt1) (get≤ lt2))

  max-sucMono : ∀ {s1 s2 s1’ s2’} → max s1 s2 ≤ max s1’ s2’ → max s1 s
  max-sucMono lt = mk≤ (indMax-sucMono (get≤ lt))


ℕLim : (ℕ → Tree) → Tree
ℕLim f = Lim Cℕ (λ cn → f (Iso.fun CℕIso cn))

max’ : Tree → Tree → Tree
max’ t1 t2 = ℕLim (λ n → if0 n t1 t2)

max’-≤L : ∀ {t1 t2} → t1 ≤ max’ t1 t2
max’-≤L {t1} {t2}
  = subst (λ x → t1 ≤ if0 x t1 t2) (sym (Iso.rightInv CℕIso 0)) ≤-refl ≤⨾
    ≤-limUpperBound (Iso.inv CℕIso 0)

max’-≤R : ∀ {t1 t2} → t2 ≤ max’ t1 t2
max’-≤R {t1} {t2}
```

```
    = subst (λ x ⟶ t2 ≤ if0 x t1 t2) (sym (Iso.rightInv CℕIso 1)) ≤-refl ≤⨾
      ≤-limUpperBound (Iso.inv CℕIso 1)

max'-Idem : ∀ {t} ⟶ max' t t ≤ t
max'-Idem {t} = ≤-limLeast helper
    where
    helper : ∀ k ⟶ if0 (Iso.fun CℕIso k) t t ≤ t
    helper k with Iso.fun CℕIso k
    ... | zero = ≤-refl
    ... | suc n = ≤-refl

max'-Mono : ∀ {t1 t2 t1' t2'}
    ⟶ t1 ≤ t1' ⟶ t2 ≤ t2'
    ⟶ max' t1 t2 ≤ max' t1' t2'
max'-Mono {t1} {t2} {t1'} {t2'} lt1 lt2 = ≤-extLim helper
    where
    helper : ∀ k ⟶ if0 (Iso.fun CℕIso k) t1 t2 ≤ if0 (Iso.fun CℕIso k) t1' t2'
    helper k with Iso.fun CℕIso k
    ... | zero = lt1
    ... | suc n = lt2

max'-LUB : ∀ {t1 t2 t} ⟶ t1 ≤ t ⟶ t2 ≤ t ⟶ max' t1 t2 ≤ t
max'-LUB lt1 lt2 = max'-Mono lt1 lt2 ≤⨾ max'-Idem

max≤max' : ∀ {t1 t2} ⟶ max t1 t2 ≤ max' t1 t2
max≤max' = max-LUB max'-≤L max'-≤R

max'≤max : ∀ {t1 t2} ⟶ max' t1 t2 ≤ max t1 t2
max'≤max = max'-LUB max-≤L max-≤R
```

References

Yves Bertot and Pierre Castéran. 2004. Interactive Theorem Proving and Program Development. Springer-Verlag.

Marc Bezem and Thierry Coquand. 2022. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. Theoretical Computer Science 913 (2022), 1–7. https://doi.org/10.1016/j.tcs.2022.01.017

Jonathan H.W. Chan. 2022. Sized dependent types via extensional type theory. Master's thesis. University of British Columbia. https://doi.org/10.14288/1.0416401

Nathan Corbyn. 2021. Proof Synthesis with Free Extensions in Intensional Type Theory. Technical Report. University of Cambridge. MEng Dissertation.

Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. 2023. Type-theoretic approaches to ordinals. Theoretical Computer Science 957 (2023), 113843. https://doi.org/10.1016/j.tcs.2023.113843

The Univalent Foundations Program. 2013. Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study.