Strictly Monotone Brouwer Trees for Well Founded Recursion Over Multiple Values

Anonymous Author(s)

Abstract

Ordinals can be used to prove the termination of dependently typed programs. Brouwer trees are a particular ordinal notation that make it very easy to assign sizes to higher order data structures. They extend unary natural numbers with a limit constructor, so a function's size can be the least upper bound of the sizes of values from its image. These can then be used to define well founded recursion: any recursive calls are allowed so long as they are on values whose sizes are strictly smaller than the current size.

Unfortunately, Brouwer trees are not algebraically well behaved. They can be characterized equationally as a join-semilattice, where the join takes the maximum of two trees. However, this join does not interact well with the successor constructor, so it does not interact properly with the strict ordering used in well founded recursion.

We present Strictly Monotone Brouwer trees (SMB-trees), a refinement of Brouwer trees that are algebraically well behaved. SMB-trees are built using functions with the same signatures as Brouwer tree constructors, and they satisfy all Brouwer tree inequalities. However, their join operator distributes over the successor, making them suited for well founded recursion or equational reasoning.

This paper teaches how, using dependent pairs and careful definitions, an ill behaved definition can be turned into a well behaved one. Our approach is axiomatically lightweight: it does not rely on Axiom K, univalence, quotient types, or Higher Inductive Types. We implement a recursively-defined maximum operator for Brouwer trees that matches on successors and handles them specifically. Then, we define SMB-trees as the subset of Brouwer trees for which the recursive maximum computes a least upper bound. Finally, we show that every Brouwer tree can be transformed into a corresponding SMB-tree by joining it with itself an infinite number of times. All definitions and theorems are implemented in Agda.

Keywords: dependent types, Brouwer trees, well founded recursion

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 are ill suited to recursion over higher-order data structures, where some fields contain functions.

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 $\uparrow$ such that max $(\uparrow t)\ t \approx \uparrow t$ and $\uparrow(\text{max}\ s\ t) = \text{max}(\uparrow s)\ (\uparrow t)$

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 [Allais et al. 2023; 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 Brouwer {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 $\mathbb{C}$ 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 $\mathbb{C}$:= 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 $\mathbb{C}$ 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)))
```

Crucially, it is also transitive, making the relation a pre-order. 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 = (≤↑t _) ≤⨾ pf

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

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

[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.

```
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 ↑t ≤ (Lim c f) is with ≤-cocone, yielding a concrete index k for which ↑ t ≤ 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
-- Symmetric

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
```

JE ►TODO update description◄ From these properties, we can compute several other useful properties: monotonicity, commutativity, and that it is in fact the least of all upper bounds.

```
limMaxMono : ∀ {t₁ t₂ t₁′ t₂′}
  → t₁ ≤ t₁′ → t₂ ≤ t₂′
  → limMax t₁ t₂ ≤ limMax t₁′ t₂′

limMaxCommut : ∀ {t₁ t₂} → limMax t₁ t₂ ≤ limMax t₂ t₁

limMaxLUB : ∀ {t₁ t₂ t} → t₁ ≤ t → t₂ ≤ t → limMax t₁ t₂ ≤ t
```

It is not surprising that this version of the maximum is a least upper bound: by definition Lim computes the least upper bound of a function's image, and limMax is simply Lim applied to a function whose image has (at most) two elements.

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 ↑t ≤ Lim c f is using the ≤-cocone constructor. So we would need to prove that ↑(max t_1 t_2) ≤ t_1' or that ↑(max t_1 t_2) ≤ t_2', which cannot be deduced from the premises alone. What we want is to have ↑max (t_1) t_2 ≤ max(↑t_1) (↑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 indMax (↑t_1) (↑t_2) = ↑(indMax t_1 t_2) hold definitionally. Then, provided indMax 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
```

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 indMax ↑Z Lim c f will not actually be an upper bound for ↑Z if c has no inhabitants. In ?? we show how to circumvent this restriction.

Under the assumption that all code are inhabited, we obtain several of our desired properties for a maximum: it is an upper bound, it is monotone and strictly monotonicity, and it is associative and commutative.

```
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
  -- Symmetric

  indMax-monoL : ∀ {t₁ t₁' t₂}
    → t₁ ≤ t₁' → indMax t₁ t₂ ≤ indMax t₁' t₂
  indMax-monoR : ∀ {t₁ t₂ t₂'}
    → t₂ ≤ t₂' → indMax t₁ t₂ ≤ indMax t₁ t₂'

  indMax-mono : ∀ {t₁ t₂ t₁' t₂'}
    → t₁ ≤ t₁' → t₂ ≤ t₂' → indMax t₁ t₂ ≤ indMax t₁' t₂'


  indMax-strictMono : ∀ {t₁ t₂ t₁' t₂'}
    → t₁ < t₁' → t₂ < t₂' → indMax t₁ t₂ < indMax t₁' t₂'
  indMax-strictMono lt1 lt2 = indMax-mono lt1 lt2


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

3.2.1 Limitation: Idempotence.

The problem with an inductive definition of the maximum is that we cannot prove that it is idempotent. Since indMax is associative and commutative, proving idempotence is equivalent to proving that it computes a true least-upper-bound.

The difficulty lies in showing that indMax (Lim $c\ f$) (Lim $c\ f$) ≤ (Lim $c\ f$). By our definition, indMax (Lim $c\ f$) (Lim $c\ f$) reduces to

$$(\mathsf{Lim}\ c\ \lambda x \to (\mathsf{Lim}\ c\ \lambda y \to \mathsf{indMax}\ (f\ x)\ (f\ y))) \leq \mathsf{Lim}\ c\ f$$

We cannot use ≤-cocone to prove this, since the left hand side is not necessarily equal to $f\ k$ for any $k : El\ c$. So the only possibility is to use ≤-limiting. Applying it twice, along with a use of commutatativity of indMax, we are left with the following goal:

$$(\forall x \to (\forall y \to \mathsf{indMax}\ (f\ x)\ (f\ y))) \leq \mathsf{Lim}\ c\ f$$

There is no a priori way to prove this goal without already having a proof that indMax is a least upper bound. But proving that was the whole point of proving idempotence! An inductive hypothesis would give that indMax $(f\ x)\ (f\ x) \leq f\ x \leq Lim\ c\ f$, but it does not apply when the arguments to indMax are not equal. Because we are working with constructive ordinals, we have no trichotomy property [?], and hence no guarantee that indMax $(f\ x)\ (f\ y)$ will be one of $f\ x$ and $f\ y$.

We now have two competing defintiions for the maximum: the limit version, which is not strictly monotone, and the inductive version, which is not actually a least upper bound. In the next section, we describe a large class of trees for which indMax is idempotent, and hence does compute a true upper bound. We then use that in ?? to create a version of ordinals whose join has the best properties of both limMax and indMax. JE ►TODO recall the algebraic definition of semilattice◄

4 Trees with a Strictly-Monotone Idempotent Join

4.1 Well-Behaved Trees

Our first step in defining an ordinal notation with a well behaved maximum is to identify a class of Brouwer trees which are well behaved with respect to the inductive maximum. As we saw in

The answer, it turns out, is more limits: if we indMax a term with itself an infinite number of times, the result will be idempotent with respect to indMax. First, we define a function to indMax a term with itself n times or a given number n:

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

To compute a tree equivalent to the infinite chain of applications indMax t (indMax t (indMax t …)), we take the limit of n applications over all n:

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

This operator has useful basic properties: it is monotone, and it computes an upper bound on is argument.

```
indMax∞-self : ∀ t → t ≤ indMax∞ t
```

```
indMax∞-mono : ∀ {t₁ t₂}
  → t₁ ≤ t₂
  → (indMax∞ t₁) ≤ (indMax∞ t₂)
```

However, the most important property we want from indMax∞ is that indMax is idempotent with respect to it. The first step to showing this is realizing that we can take the maximum of t and indMax∞ t and we have a tree that is no larger than indMax∞ t: because it is already an infinite chain of applications, adding one more makes no difference.

```
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)))
      (≤-refl _))
```

If adding one more indMax t has no effect, then adding n more will also have no effect:

```
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 =
  indMax-monoR (indMax-commut (nindMax t n) t)
  ≤⨾ indMax-assocL (indMax∞ t) t (nindMax t n)
  ≤⨾ indMax-monoL (indMax-∞lt1 t)
  ≤⨾ indMax-∞ltn n t
```

By our inductive definition of indMax, we have that

$$\mathsf{indMax}\ (\mathsf{indMax}\infty\ t)(\mathsf{indMax}\infty\ t)$$

is equal to

$$\mathsf{\mathbb{N}Lim}\ (\lambda n \to \mathsf{indMax}\ (\mathsf{nIndMax}\ n\ t)\ (\mathsf{indMax}\infty\ t))$$

Our previous lemma gives that, for any n, indMax∞ t is an upper bound for indMax (nIndMax n t) (indMax∞ t)). So ≤-limiting gives that the limit over all n is also bounded by indMax∞ t, i.e. Lim constructs the least of all upper bounds. This gives us our key result: up to ≤, indMax is idempotent on values constructed with indMax∞.

```
indMax∞-idem : ∀ t
  → indMax (indMax∞ t) (indMax∞ t) ≤ indMax∞ t
indMax∞-idem t =
  ≤-limiting _ λ k →
    (indMax-commut (nindMax t (Iso.fun CℕIso k)) (indMax∞ t))
  ≤⨾ indMax-∞ltn (Iso.fun CℕIso k) t
```

There is one last property to prove that will be useful in the next section: indMax∞ t is a lower bound on t, and hence equivalent to it, whenever indMax is idempotent on t. If taking indMax of t with itself does not increase it size, doing so n times will not increase it size, so again the result follows from Lim being the least upper bound.

```
indMax∞-≤ : ∀ {t} → indMax t t ≤ t → indMax∞ t ≤ t
indMax∞-≤ lt = ≤-limiting _ λ k → nindMax-≤ (Iso.fun CℕIso k) lt
  where
    nindMax-≤ : ∀ {t} n → indMax t t ≤ t → nindMax t n ≤ t
    nindMax-≤ ℕ.zero lt = ≤-Z
    nindMax-≤ {t = t} (ℕ.suc n) lt = (indMax-monoL {t₁ = nindMax t n} {t₂ = t} (nindMax-≤ n lt)) ≤⨾ lt
```

An immediate corollary of this is that indMax∞ (indMax∞ t) is equivalent to indMax∞ t.

4.2 Strictly Monotone Brouwer Trees

Now that we have identified a substantial class of well behaved Brouwer trees, we want to define a new type containing only those trees. These are SMB-trees: strictly monotone Brouwer trees. In this section, we will define them, and show how they can be given a similar interface to Brouwer trees.

To begin, we declare a new Agda module, with the same parameters we have been working with thus far: a type of codes, interpretations of those codes into types, and a code whose interpretation is isomorphic to ℕ.

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

We import all of our definitions so far, using the "Brouwer" prefix to distinguish them from the trees and ordering we are about to define. Critically, we do not instantiate these with the same interpretation function. Instead, we interpret each code wrapped in Maybe. This ensures that we always take Brouwer limits over non-empty sets, an assumption that was critical for the definitions of **??**. However, we place no such restriction on SMB-trees.

```
import Brouwer ℂ (λ c → Maybe (El c)) Cℕ (maybeNatIso CℕIso) as Brouwer
```

4.2.1 Refining Brouwer Trees.

We define SMB-trees as a dependent record, containing an underlying Brouwer tree, and a proof that indMax is idempotent on this tree.

```
record SMBTree : Set ℓ where
  constructor MkTree
  field
    rawTree : Brouwer.Tree
    isIdem : (indMax rawTree rawTree) Brouwer.≤ rawTree
open SMBTree
```

We can then define so-called "smart-constructors" corresponding to each of the constructors for Brouwer-trees: zero, successor, and limit. Zero and successor directly correspond to the Brouwer tree zero and successor. Their proofs of idempotence are trivial from the properties of Brouwer ≤.

```
opaque
  unfolding indMax

  Z : SMBTree
  Z = MkTree Brouwer.Z Brouwer.≤-Z

  ↑ : SMBTree → SMBTree
  ↑ (MkTree t pf) = MkTree (Brouwer.↑ t) (Brouwer.≤-sucMono pf)
```

However, constructing the limit of a sequence of SMB-trees is not so easy. Since we instantiated El to wrap its result in Maybe, we need to handle *nothing* for each limit, but we can use Z as a default value, since adding it to any sequence does not change the least upper bound. More challenging is how, as we saw in **??**, Brouwer trees do not have indMax (Lim c f) (Lim c f) ≤ Lim c f, so we cannot directly produce a proof of idempotence.

Our key insight is to define limits of SMB-trees using indMax∞ on the underlying trees: for any function producing SMB-trees, we take the limit of the underlying trees, then indMax that result with itself an infinite numer of times. The idempotence proof is then the property of indMax∞ that we proved in **??**.

```
Lim : ∀ (c : ℂ) → (f : El c → SMBTree) → SMBTree
Lim c f =
  MkTree
  (indMax∞ (Brouwer.Lim c (maybe′ (λ x → rawTree (f x)) Brouwer.Z)))
  (indMax∞-idem _)
```

4.2.2 Ordering SMB-trees.

SMB-trees are ordered by the order on their underlying Brouwer trees:

```
record _≤_ (t₁ t₂ : SMBTree) : Set ℓ where
  constructor mk≤
  inductive
  field
    get≤ : (rawTree t₁) Brouwer.≤ (rawTree t₂)
open _≤_
```

Having a successor function allows us to define a strict ording on SMB-trees.

```
_<_ : SMBTree → SMBTree → Set ℓ
_<_ t₁ t₂ = (↑ t₁) ≤ t₂
```

The next step is to prove that our SMB-tree constructors satisfy the same inequalities as Brouwer trees. Since SMB-trees are ordered by their underlying Brouwer trees, most properties can be directly lifted from Brouwer trees to SMB-trees.

```
opaque
  unfolding Z ↑
  ≤↑ : ∀ t → t ≤ ↑ t
  ≤↑ t = mk≤ (Brouwer.≤↑t _)

  _≤⨾_ : ∀ {t₁ t₂ t3} → t₁ ≤ t₂ → t₂ ≤ t3 → t₁ ≤ t3
  _≤⨾_ (mk≤ lt1) (mk≤ lt2) = mk≤ (Brouwer.≤-trans lt1 lt2)

  ≤-refl : ∀ {t} → t ≤ t
  ≤-refl = mk≤ (Brouwer.≤-refl _)
```

The constructors for $\leq$ each have a counterpart for SMB-trees. For zero and successor, these are trivially lifted.

```
≤-Z : ∀ {t} → Z ≤ t
≤-Z = mk≤ Brouwer.≤-Z

≤-sucMono : ∀ {t₁ t₂} → t₁ ≤ t₂ → ↑ t₁ ≤ ↑ t₂
≤-sucMono (mk≤ lt) = mk≤ (Brouwer.≤-sucMono lt)
```

The constructors for ordering limits require more attention. To show that an SMB-tree limit is an upper-bound, we use the fact that the underlying limit was an upper bound, and the fact that indMax∞ is an upper bound, since the SMB-tree Lim wraps its result in indMax∞. Note that, since we already have transitivity for our new $\leq$, we can simply show that $f\ k$ is less than the limit of f, avoiding the more complicated form of ≤-cocone.

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

Finally, we need to show that the SMT-tree limit is less than all other upper bounds. Suppose t : SMBTRee is an upper bound for f, and t_u is the underlying tree for t, and f_u computes the underlying trees for f. Then ≤-limiting gives that the underlying tree for t is an upper bound for the trees underlying the image of f. However, the SMB-tree limit wraps its result in indMax∞. The monotonicity of indMax∞ then gives that indMax(Lim $c\ f_u$) is less than indMax∞ t'. In **??**, we showed that indMax∞ had no effect on Brouwer trees that indMax was idempotent on. This is exactly what the isIdem field of SMB-trees contains! So we have indMax∞ $t' \leq t'$, and transitivity gives our result.

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

4.2.3 The Join for SMB-trees.

Our whole reason for defining SMB-trees was to define a well-behaved maximum operator, and we finally have the tools to do so. We can define the join in terms of indMax on the underlying trees. The proof that the indMax is idempotent on the result follows from associativity, commutativity, and monotonicity of indMax.

```
opaque
  unfolding indMax Z ↑ indMaxView
  max : SMBTree → SMBTree → SMBTree
  max t₁ t₂ =
    MkTree
      (indMax (rawTree t₁) (rawTree t₂))
      (indMax-swap4
        Brouwer.≤ ⨾ indMax-mono (isIdem t₁) (isIdem t₂))
```

For Brouwer trees, indMax had all the properties we wanted except for idempotence. All of these can be lifted directly to SMB-trees:

```
max-≤L : ∀ {t₁ t₂} → t₁ ≤ max t₁ t₂

max-≤R : ∀ {t₁ t₂} → t₂ ≤ max t₁ t₂

max-mono : ∀ {t₁ t₁′ t₂ t₂′} → t₁ ≤ t₁′ → t₂ ≤ t₂′ →
  max t₁ t₂ ≤ max t₁′ t₂′

max-idem≤ : ∀ {t} → t ≤ max t t

max-commut : ∀ t₁ t₂ → max t₁ t₂ ≤ max t₂ t₁

max-assocL : ∀ t₁ t₂ t3 → max t₁ (max t₂ t3) ≤ max (max t₁ t₂) t3

max-assocR : ∀ t₁ t₂ t3 → max (max t₁ t₂) t3 ≤ max t₁ (max t₂ t3)
```

In particular, max is strictly monotone, and distributes over the successor:

```
max-strictMono : ∀ {t₁ t₁′ t₂ t₂′ : SMBTree}
  → t₁ < t₁′ → t₂ < t₂′ → max t₁ t₂ < max t₁′ t₂′

max-sucMono : ∀ {t₁ t₂ t₁′ t₂′ : SMBTree}
  → max t₁ t₂ ≤ max t₁′ t₂′ → max t₁ t₂ < max (↑ t₁′) (↑ t₂′)
```

However, because we restricted SMB-trees to only contain Brouwer trees that indMax is idempotent on, we can prove that Max is idempotent for SMB-trees:

```
max-idem : ∀ {t : SMBTree} → max t t ≤ t
max-idem {t = MkTree t pf} = mk≤ pf
```

These together are enough to prove that our maximum is the least of all upper bounds.

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

Perhaps surprisingly, this means that an SMB-tree version of limMax is equivalent to max, since they are both the least upper bound:

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

max' : SMBTree → SMBTree → SMBTree
max' t₁ t₂ = ℕLim (λ n → if0 n t₁ t₂)

max'-≤L : ∀ {t₁ t₂} → t₁ ≤ max' t₁ t₂

max'-≤R : ∀ {t₁ t₂} → t₂ ≤ max' t₁ t₂

max'-LUB : ∀ {t₁ t₂ t} → t₁ ≤ t → t₂ ≤ t → max' t₁ t₂ ≤ t

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
```

4.2.4 Well Founded Ordering on SMB-trees. Our motivation for defining SMB-trees was defining well founded recursion, so the final piece of our definition is a proof that the strict ordering of SMB-trees is well founded. Intuitively this should hold: if there are no infinite descending chains of Brouwer trees, and there are fewer SMB-trees than Brouwer trees, then there can be no infinite descending chains of SMB-trees. The key lemma is that an SMB-tree is accessible if its underlying Brouwer tree is.

```
sizeWF : WellFounded _<_
sizeWF t = sizeAcc (Brouwer.ordWF (rawTree t))
  where
    sizeAcc : ∀ {t}
      → Acc Brouwer._<_ (rawTree t)
      → Acc _<_ t
    sizeAcc {t} (acc x)
      = acc ((λ y lt → sizeAcc (x (rawTree y) (get≤ lt))))
```

Thus, we have an ordinal type with limits, a strictly monotone join, and well founded recursion.

References

Guillaume Allais, Edwin Brady, Nathan Corbyn, Ohad Kammar, and Jeremy Yallop. 2023. Frex: dependently-typed algebraic simplification. arXiv:2306.15375 [cs.PL]

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.